Luong Nguyen | S3927460

Embedded System: OS and Interfacing

EEET2490

Prof. Linh Tran Duc

RMIT UNIVERSITY

ADDITIONAL FEATURES FOR BARE METAL OS

Assessment 2 – Individual Assignment Report

Table of Contents

I. INTRODUCTION

In the field of computer engineering, embedded systems and their operating environments continues to be a critical area of study and development. This report will focus on the practical aspects of the development of DoorOS – a bare-metal operating system, designed to enhance both theoretical knowledge and practical skills for embedded systems programming. The objective of this project is to implement a set of additional features to our existing bare-metal OS to improve its functionality and user interaction capabilities.

The additional features discussed in this report include the development of a Command Line Interpreter (CLI), and improvements to the UART driver. Additionally, will be discussing some common types of sensors to demonstrate real-world interfacing. Each feature aims to extend the OS usability in embedded applications, reflecting common tasks that a system might handle in an operational setting.

This assignment not only provides hands-on experience with the fundamental compenents of system programming but also integrates advanced features typically found in more complex operating systems. Through the task outlined in this assignment, the report will demonstrate the application of these featrures in a simplified environment, helping to connect the gap between theoretical concepts and practical applications in embedded systems development.

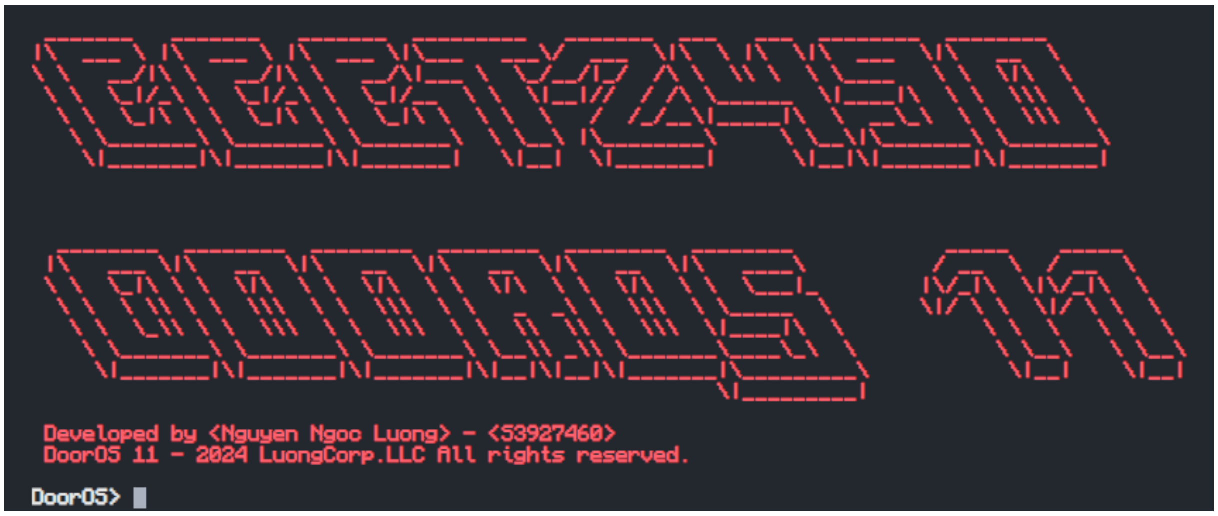
By the end of this report, the enhancements made to the OS will be fully documented, including the methodology used in the implementation process, testing and demonstration on both emulated environments using QEMU and actual hardware using Raspberry Pi 4, and an analysis of the challenges and learning encountered throughout the process.

II. ADDITIONAL FEATURES FOR BARE METAL OS

1. Welcome message and Command Line Interpreter (CLI)

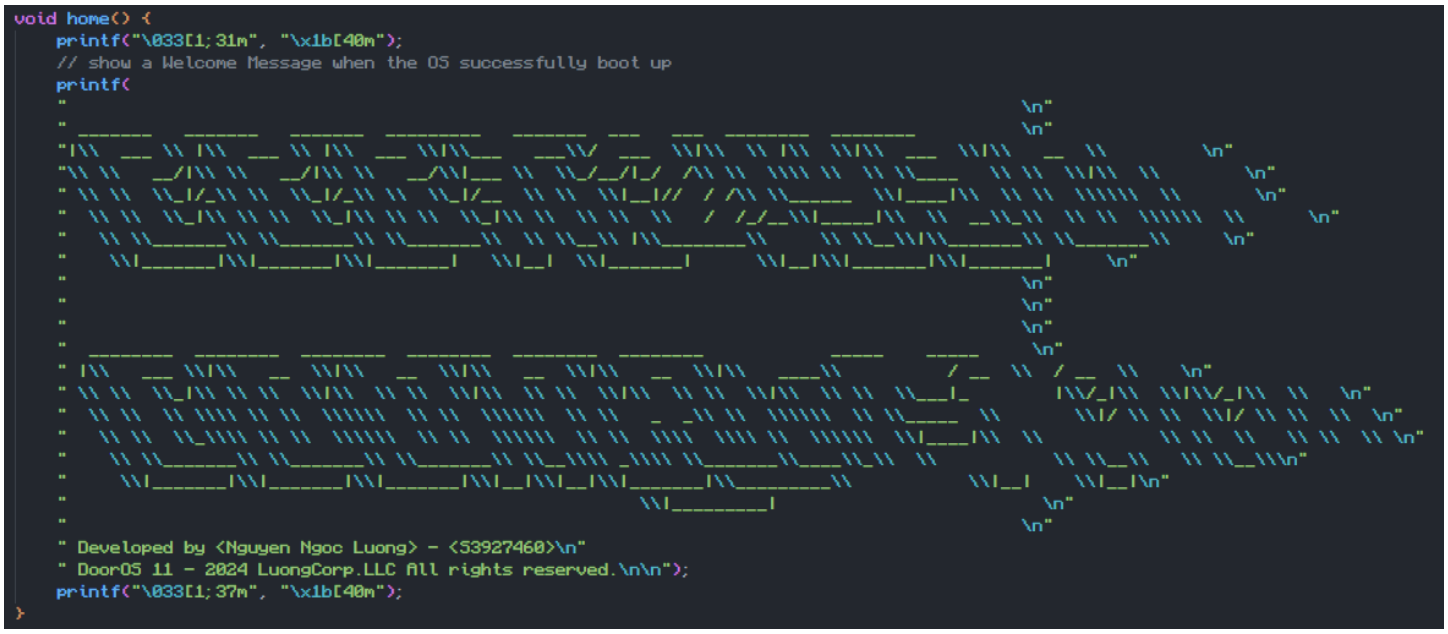
Welcome message implementation

Upon booting up the program, DoorOS will greets the user with a distinctive welcome message displayed in ASCII art, this feature not only enhances the user experience but also provides a visual confirmation that the system has launched successfully.

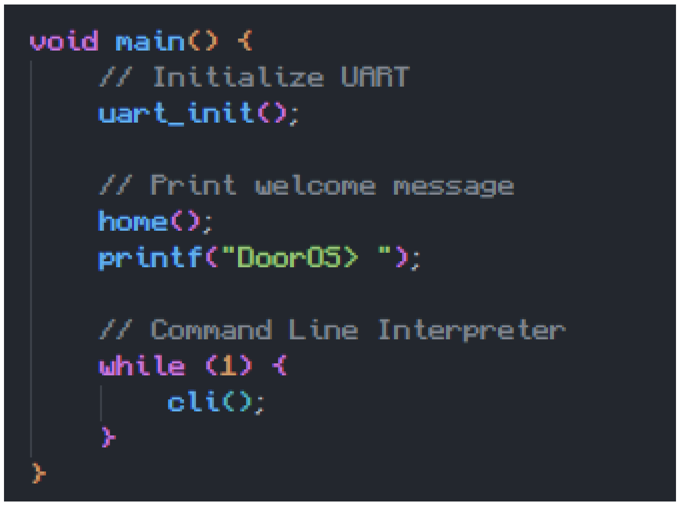


```
Developed by <Nguyen Ngoc Luong> - <S3927460>
DoorOS 11 - 2024 LuongCorp.LLC All rights reserved.

DoorOS>
```

Using the provided tools to convert Text to ASCII art, I converted the welcome message into an ASCII art string, I then utilized a custom *printf()* function to display the message to the terminal.



```
void home() {
    printf("\033[1;31m", "\x1b[40m");
    // show a Welcome Message when the OS successfully boot up
    printf(
    "                                                                              \n"
    "  _______  _______  _______  _________  _______  ___   ___  ________  ________                    \n"
    "|\\  ___ \\ |\\  ___ \\ |\\  ___ \\|\\___   ___\\/  ___  \\|\\  \\ |\\  \\|\\  ___  \\|\\  ___  \\        \n"
    "\\ \\   __/|\\ \\   __/|\\ \\   __/\\|___ \\  \\_|\\/__/|_/  /\\ \\  \\\\\ \\  \\ \\  \\____   \\ \\  \\|\\  \\  \\          \n"
    " \\ \\  \\_|/_\\ \\  \\_|/_\\ \\  \\_|/__  \\ \\  \\\\|__|//  / /\\ \\  \\_____  \\\|____|\\  \\  \\ \\  \\\\\\\\  \\        \n"
    "  \\ \\  \\_|\\ \\ \\  \\_|\\ \\ \\  \\_|\\ \\  \\ \\  \\   \\   /  /_/__\\\|____|\\  \\  __\\_\\  \\ \\  \\\\\\\\  \\           \n"
    "   \\ \\_______\\ \\_______\\ \\_______\\  \\ \\__\\ |\\________\\     \\ \\__\\\|\\_______\\ \\_______\\      \n"
    "    \\|_______|\\|_______|\\|_______|   \\|__|  \\|_______|      \\|__|\\|_______|\\|_______|     \n"
    "                                                                              \n"
    "                                                                              \n"
    "                                                                              \n"
    "   ________  ________  ________  ________  ________  ________        _____      _____     \n"
    " |\\   ___ \\\|\\   __  \\\|\\   __  \\\|\\   __  \\\|\\   __  \\\|\\   ____\\          / __  \\  / __  \\     \n"
    " \\ \\  \\_|\\ \\ \\  \\\|\\  \\ \\  \\\|\\  \\ \\  \\\|\\  \\ \\  \\\|\\  \\ \\  \\___|_         |\\/_|\\  \\\|\\/_|\\  \\    \n"
    "  \\ \\  \\ \\\\ \\ \\  \\\\\\  \\ \\  \\\\\\  \\ \\   _  _\\ \\  \\\\\\  \\ \\_____  \\        \\|/ \\ \\  \\|/ \\ \\  \\   \\  \n"
    "   \\ \\  \\_\\\\ \\ \\  \\\\\\  \\ \\  \\\\\\  \\ \\  \\\\  \\\\ \\  \\\\\\  \\|____|\\  \\            \\ \\  \\   \\ \\  \\  \\ \n"
    "    \\ \\_______\\ \\_______\\ \\_______\\ \\__\\\\ _\\\\ \\_______\\____\\_\\  \\            \\ \\__\\   \\ \\__\\\\n"
    "     \\|_______|\\|_______|\\|_______|\\|__|\\|__|\\|_______|\\_________\\            \\|__|    \\|__|\n"
    "                                                   \\|_________|                \n"
    "                                                                              \n"
    " Developed by <Nguyen Ngoc Luong> - <S3927460>\n"
    " DoorOS 11 - 2024 LuongCorp.LLC All rights reserved.\n\n");
    printf("\033[1;37m", "\x1b[40m");
}
```



```
void main() {
    // Initialize UART
    uart_init();

    // Print welcome message
    home();
    printf("DoorOS> ");

    // Command Line Interpreter
    while (1) {
        cli();
    }
}
```

In the main function, after initializing UART, the *home()* function is called to display the welcome message, this also act as a home screen where the user could return to by typing ***home*** in the terminal.

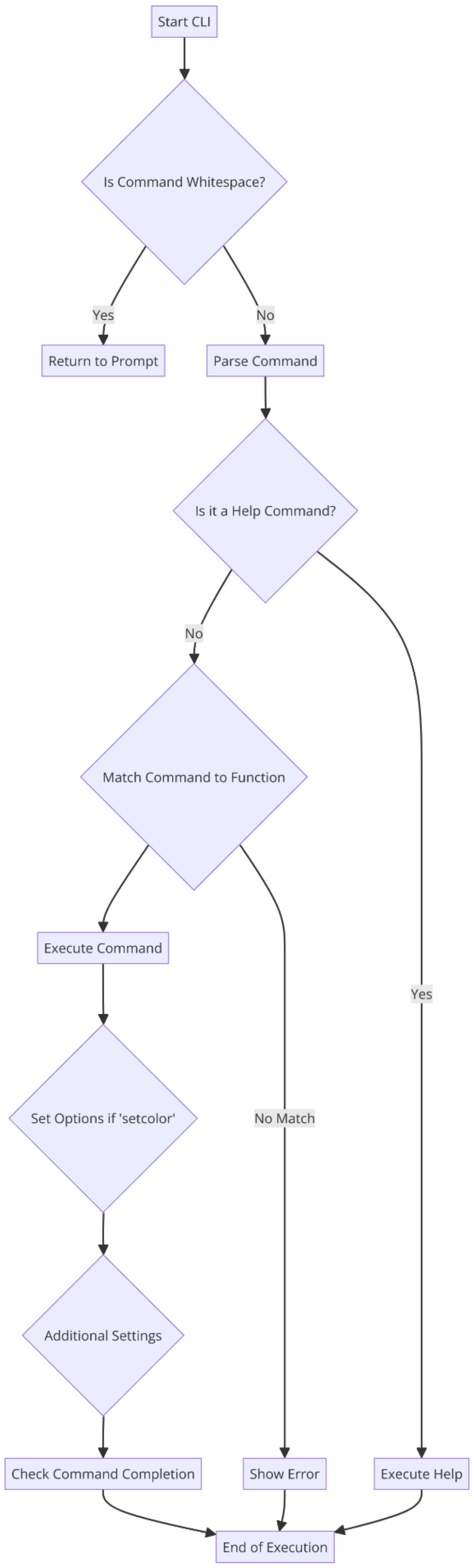
Command Line Interpreter (CLI)

The command Line Interpreter (CLI) is an essential interface in many OS, expecially in the realm of embedded systems where graphical user interfaces (GUIs) may be impractical or resource-intensive. A CLI allows user to type text command to a terminal, where the command will the be executed by the OS, this method of interaction proivide a flexible, and scriptable way of controlling a computer system[1].

In the context of DoorOS development, the CLI Is not just a feature but a fundamental component that facilitates direct communication between the user and the OS.

CLI Requirements

The CLI for DoorOS includes several functionalities aimed at enhancing usability and effectiveness. Each feature is integrated with the intention of making the system more accessible and easier to use:



Welcome Message: Displays a custom ASCII art welcome message upon system boot-up, immediately indicating that the system is operational and ready to accept commands.

Command Prompt: Shows a consistent command prompt (e.g., DoorOS>) where users can type commands. This prompt reflects the current state of the system and provides a ready indicator for user input.

Command Execution: Ability to read user input and execute a variety of built-in commands, translating user requests into actions performed by the operating system.

Basic Commands:

help: Displays a list of all available commands or detailed information about a specific command when requested.

clear: Clears the screen to keep the CLI tidy and focused on current tasks.

setcolor: Changes the text and/or background color of the CLI for better visibility or user preference.

Advanced Commands:

showinfo: Provides detailed information about the system, such as board revision and MAC address, in a clear and understandable format.

Auto-completion: Enhances typing efficiency by allowing users to complete commands by pressing the TAB key.

Command History: Enables users to navigate through a history of previously entered commands using designated keys, improving efficiency in repeated tasks.

Error Handling: Ensures that any errors or invalid commands are handled gracefully, providing users with helpful error messages and guidance on correct command usage.

Customizability: Offers features like ***setcolor*** that allow users to personalize their CLI experience, adapting the interface to suit their preferences or needs.

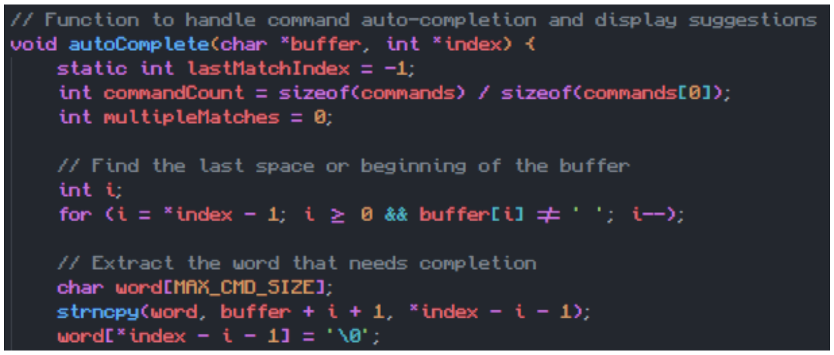
CLI Implementation

Auto-completion

The autoComplete function helps users complete commands more quickly by suggesting and automatically filling in command names based on partial inputs. It's triggered when the user starts typing a command and pressed the TAB key to initiate auto-completion.

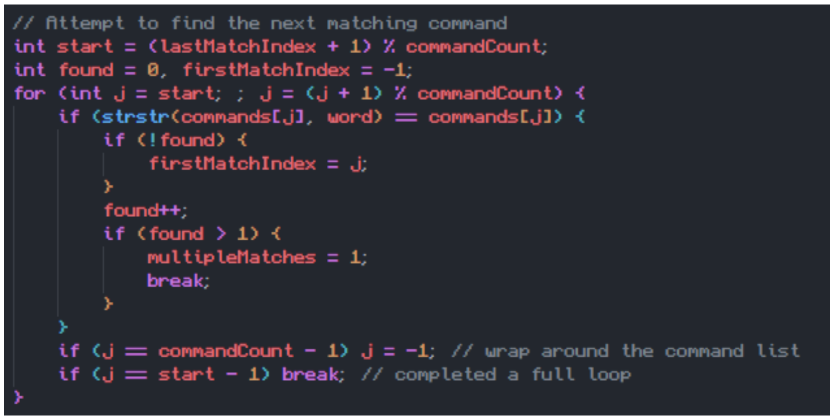
Identify the last word typed: The function first determines where the last word in the command buffer starts. It does this by searching backward from the current cursor position (*index) until it finds a space (indicating the start of the last word) or reaches the beginning of the buffer.

Extract the current word: The word that needs to be completed is extracted from the buffer and stored in word. This extraction is based on the position of the last space found, or from the start of the buffer if no space was found.



```
// Function to handle command auto-completion and display suggestions
void autoComplete(char *buffer, int *index) {
    static int lastMatchIndex = -1;
    int commandCount = sizeof(commands) / sizeof(commands[0]);
    int multipleMatches = 0;

    // Find the last space or beginning of the buffer
    int i;
    for (i = *index - 1; i >= 0 && buffer[i] != ' '; i--);

    // Extract the word that needs completion
    char word[MAX_CMD_SIZE];
    strncpy(word, buffer + i + 1, *index - i - 1);
    word[*index - i - 1] = '\0';
```

Search for matches: The function then searches through a list of available commands (commands) to find matches that begin with the extracted word. The search starts from the position right after the last successful match (to facilitate cycling through multiple matches with repeated presses of the auto-complete key).



```
// Attempt to find the next matching command
int start = (lastMatchIndex + 1) % commandCount;
int found = 0, firstMatchIndex = -1;
for (int j = start; ; j = (j + 1) % commandCount) {
    if (strstr(commands[j], word) == commands[j]) {
        if (!found) {
            firstMatchIndex = j;
        }
        found++;
        if (found > 1) {
            multipleMatches = 1;
            break;
        }
    }
    if (j == commandCount - 1) j = -1; // wrap around the command list
    if (j == start - 1) break; // completed a full loop
}
```

Handle multiple matches:

If multiple commands match the partially typed word, the function lists all possible completions to inform the user of available options.

The buffer is updated with the first matching command, allowing the user to either accept this auto-completion by continuing to type or trigger the auto-completion again to cycle through other matches.

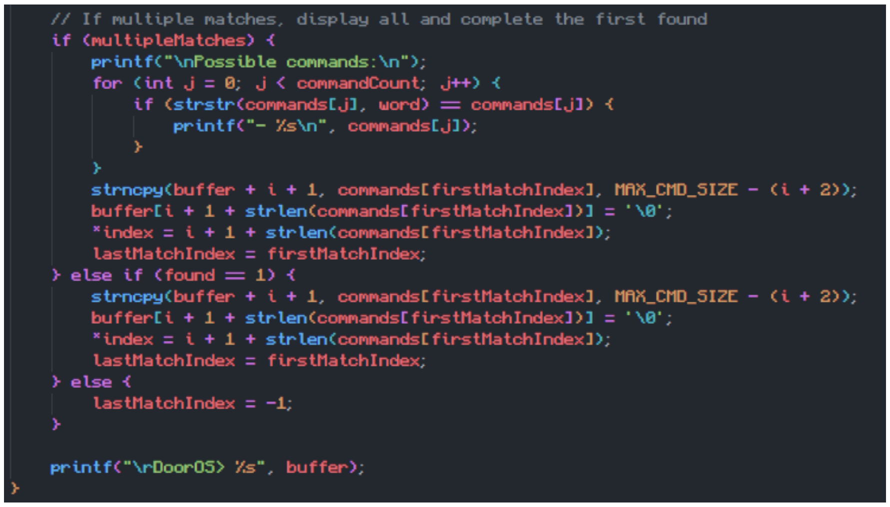
Handle a single match: If only one command matches, the buffer is directly updated with this command, effectively completing the command for the user.

Update cursor position: After updating the buffer with a command, the cursor index is adjusted to reflect the new position at the end of the completed command.

Reset or update the last match index:

If no matches are found, lastMatchIndex is reset to -1, indicating that the next attempt to auto-complete should start from the beginning of the command list.

If a match is found, lastMatchIndex is updated to the index of the first match, ensuring that subsequent uses of the auto-complete function can cycle through other potential matches if the user continues to invoke auto-completion.



```c
    // If multiple matches, display all and complete the first found
    if (multipleMatches) {
        printf("\nPossible commands:\n");
        for (int j = 0; j < commandCount; j++) {
            if (strstr(commands[j], word) == commands[j]) {
                printf("- %s\n", commands[j]);
            }
        }
        strncpy(buffer + i + 1, commands[firstMatchIndex], MAX_CMD_SIZE - (i + 2));
        buffer[i + 1 + strlen(commands[firstMatchIndex])] = '\0';
        *index = i + 1 + strlen(commands[firstMatchIndex]);
        lastMatchIndex = firstMatchIndex;
    } else if (found == 1) {
        strncpy(buffer + i + 1, commands[firstMatchIndex], MAX_CMD_SIZE - (i + 2));
        buffer[i + 1 + strlen(commands[firstMatchIndex])] = '\0';
        *index = i + 1 + strlen(commands[firstMatchIndex]);
        lastMatchIndex = firstMatchIndex;
    } else {
        lastMatchIndex = -1;
    }

    printf("\rDoorOS> %s", buffer);
}
```

Display the updated buffer: Finally, the updated command line, including any auto-completed command, is printed back to the terminal with the prompt "DoorOS>". This gives the user immediate feedback on the result of the auto-completion and allows them to continue typing or executing commands.

Terminal Demonstration:

As this feature is not suitable for image demonstration as it require live button input, I will include it in the demonstration video.

Command History

The navigateCommandHistory function enables users to move through the command history using designated keys. In the code, the keys '+' and '_' are used to navigate up and down through the history, respectively.

Directional Navigation:

Up Navigation ('+'): When the user presses '+', the function attempts to move up in the history (i.e., towards older commands). If it's not already at the oldest command, it decrements the CMD_TRACKER_INDEX, which points to the current command in the command history array CMD_TRACKER.

Down Navigation ('_'): Conversely, when the user presses '_', the function moves down in the history (i.e., towards newer commands). If the end of the command history is reached, it clears the CLI buffer to indicate no older commands are available.

Buffer Management: The command corresponding to the new CMD_TRACKER_INDEX is copied into the CLI buffer, replacing what was previously there. This allows the user to edit or execute it as if it were freshly typed.

Edge Cases:

At the newest command: If moving up in history and the newest command is reached (i.e., CMD_TRACKER_INDEX == 20), the CLI buffer is cleared to signal that there are no more recent commands to view.

History Access Control: The variable accessHistory manages entry into the history navigation mode. If the history navigation key is pressed (+ or _), the function sets accessHistory to 1, indicating that the user is navigating the history. This prevents inadvertent changes to the CMD_TRACKER_INDEX when other keys are pressed.

CLI Buffer Index Update: After updating the CLI buffer with a command from the history, the *index is set to the length of the new command in the buffer, positioning the cursor at the end of the command for further editing or execution.\

Terminal Demonstration:

As this feature is not suitable for image demonstration as it require live button input, I will include it in the demonstration video.

Set text and background color

The setcolor feature allows users to customize the text and background colors of their terminal output. This customization is achieved through ANSI escape codes, which are sequences of characters that control the appearance of the terminal display. Let's delve into how this feature works, its components, and its significance.

ANSI Escape Codes

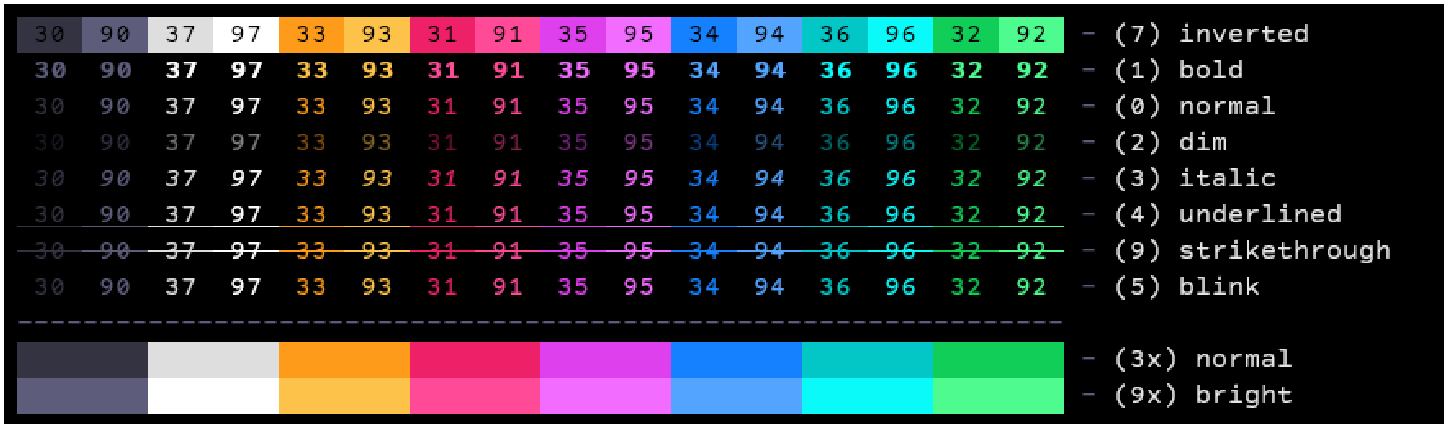
ANSI escape codes start with an escape character followed by a sequence of characters that specifies the color or format. For text colors, the codes might look like *\033[1;30m* where:

- \033 is the escape character.
- [1;30m specifies the style and color (in this case, black).

For background colors, the codes are similar but use different numbers, like *\x1b[44m* for a blue background:

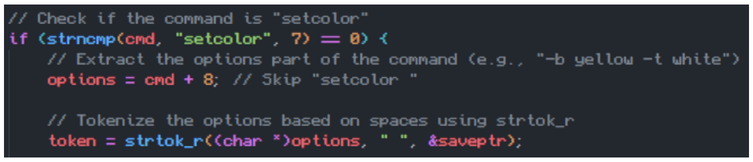
- \x1b is another way to write the escape character.

- [44m specifies the background color blue.



```
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (7) inverted
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (1) bold
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (0) normal
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (2) dim
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (3) italic
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (4) underlined
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (9) strikethrough
30 90 37 97 33 93 31 91 35 95 34 94 36 96 32 92 - (5) blink
-----------------------------------------------
                                                 - (3x) normal
                                                 - (9x) bright
```

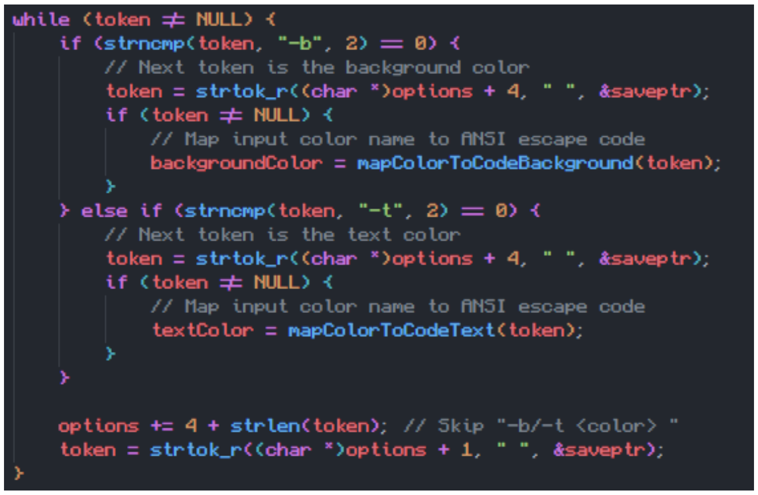
Command Parsing:



```c
// Check if the command is "setcolor"
if (strncmp(cmd, "setcolor", 7) == 0) {
    // Extract the options part of the command (e.g., "-b yellow -t white")
    options = cmd + 8; // Skip "setcolor "

    // Tokenize the options based on spaces using strtok_r
    token = strtok_r((char *)options, " ", &saveptr);
```

- The function checks if the entered command starts with setcolor.
- If it does, the command extracts the options for text and background colors that follow. These options are parsed using the strtok_r function, which tokenizes the input string based on spaces.

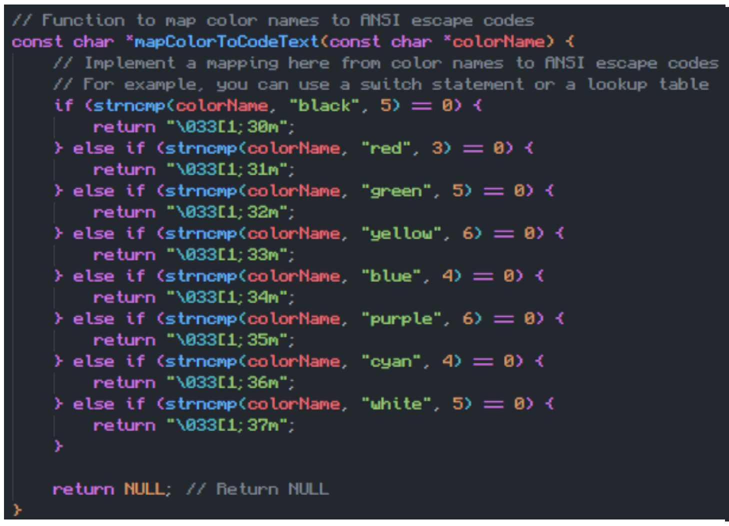
Option Handling:



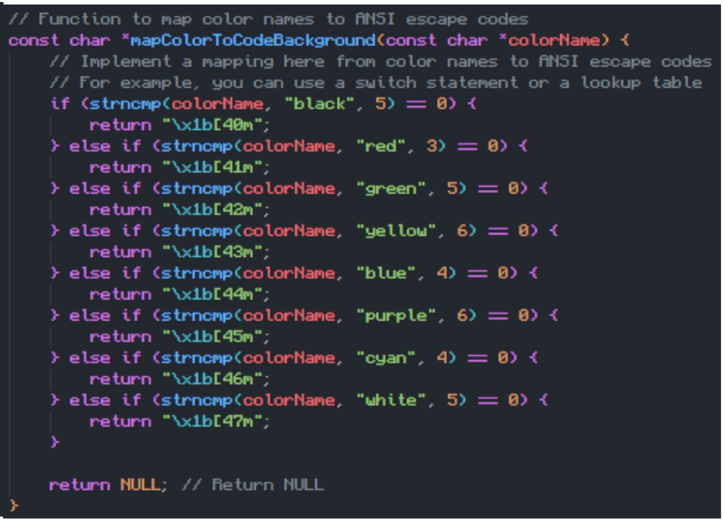
```c
while (token != NULL) {
    if (strncmp(token, "-b", 2) == 0) {
        // Next token is the background color
        token = strtok_r((char *)options + 4, " ", &saveptr);
        if (token != NULL) {
            // Map input color name to ANSI escape code
            backgroundColor = mapColorToCodeBackground(token);
        }
    } else if (strncmp(token, "-t", 2) == 0) {
        // Next token is the text color
        token = strtok_r((char *)options + 4, " ", &saveptr);
        if (token != NULL) {
            // Map input color name to ANSI escape code
            textColor = mapColorToCodeText(token);
        }
    }

    options += 4 + strlen(token); // Skip "-b/-t <color> "
    token = strtok_r((char *)options + 1, " ", &saveptr);
}
```

- For each token that matches -t or -b, the function fetches the next token, which represents the color.
- The color token is then passed to mapping functions that convert the color names to their corresponding ANSI escape codes.

Color Mapping:



```c
// Function to map color names to ANSI escape codes
const char *mapColorToCodeText(const char *colorName) {
    // Implement a mapping here from color names to ANSI escape codes
    // For example, you can use a switch statement or a lookup table
    if (strncmp(colorName, "black", 5) == 0) {
        return "\033[1;30m";
    } else if (strncmp(colorName, "red", 3) == 0) {
        return "\033[1;31m";
    } else if (strncmp(colorName, "green", 5) == 0) {
        return "\033[1;32m";
    } else if (strncmp(colorName, "yellow", 6) == 0) {
        return "\033[1;33m";
    } else if (strncmp(colorName, "blue", 4) == 0) {
        return "\033[1;34m";
    } else if (strncmp(colorName, "purple", 6) == 0) {
        return "\033[1;35m";
    } else if (strncmp(colorName, "cyan", 4) == 0) {
        return "\033[1;36m";
    } else if (strncmp(colorName, "white", 5) == 0) {
        return "\033[1;37m";
    }

    return NULL; // Return NULL
}
```



```c
// Function to map color names to ANSI escape codes
const char *mapColorToCodeBackground(const char *colorName) {
    // Implement a mapping here from color names to ANSI escape codes
    // For example, you can use a switch statement or a lookup table
    if (strncmp(colorName, "black", 5) == 0) {
        return "\x1b[40m";
    } else if (strncmp(colorName, "red", 3) == 0) {
        return "\x1b[41m";
    } else if (strncmp(colorName, "green", 5) == 0) {
        return "\x1b[42m";
    } else if (strncmp(colorName, "yellow", 6) == 0) {
        return "\x1b[43m";
    } else if (strncmp(colorName, "blue", 4) == 0) {
        return "\x1b[44m";
    } else if (strncmp(colorName, "purple", 6) == 0) {
        return "\x1b[45m";
    } else if (strncmp(colorName, "cyan", 4) == 0) {
        return "\x1b[46m";
    } else if (strncmp(colorName, "white", 5) == 0) {
        return "\x1b[47m";
    }

    return NULL; // Return NULL
}
```

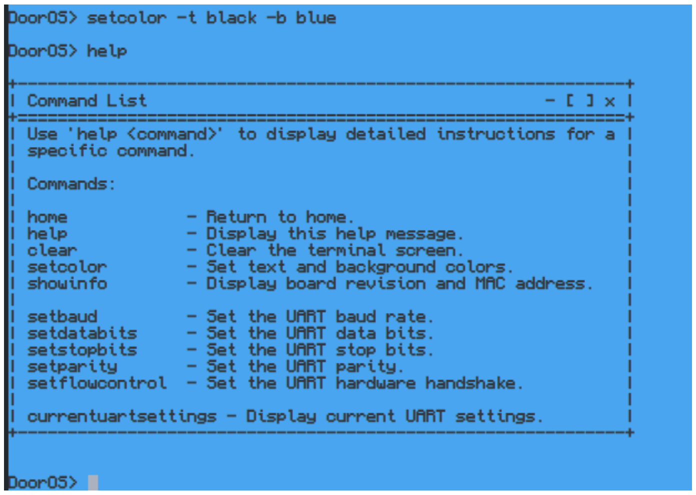
- mapColorToCodeText and mapColorToCodeBackground are two functions used to map the textual color names to ANSI escape codes. These functions use simple conditional checks (if or else if) to match the color name to the appropriate ANSI code.
- Each color name (i.e "black", "red", "blue") is associated with a unique ANSI code that instructs the terminal to change the color accordingly.

Applying the Color:

After the appropriate ANSI codes are retrieved, they are applied to the terminal. This is typically done by printing these ANSI codes directly to the terminal, where the terminal interprets them to change the display settings.

Terminal Demonstration:

Here I have set the text color to black and background color to white using the syntax ***setcolor -t black -b white***



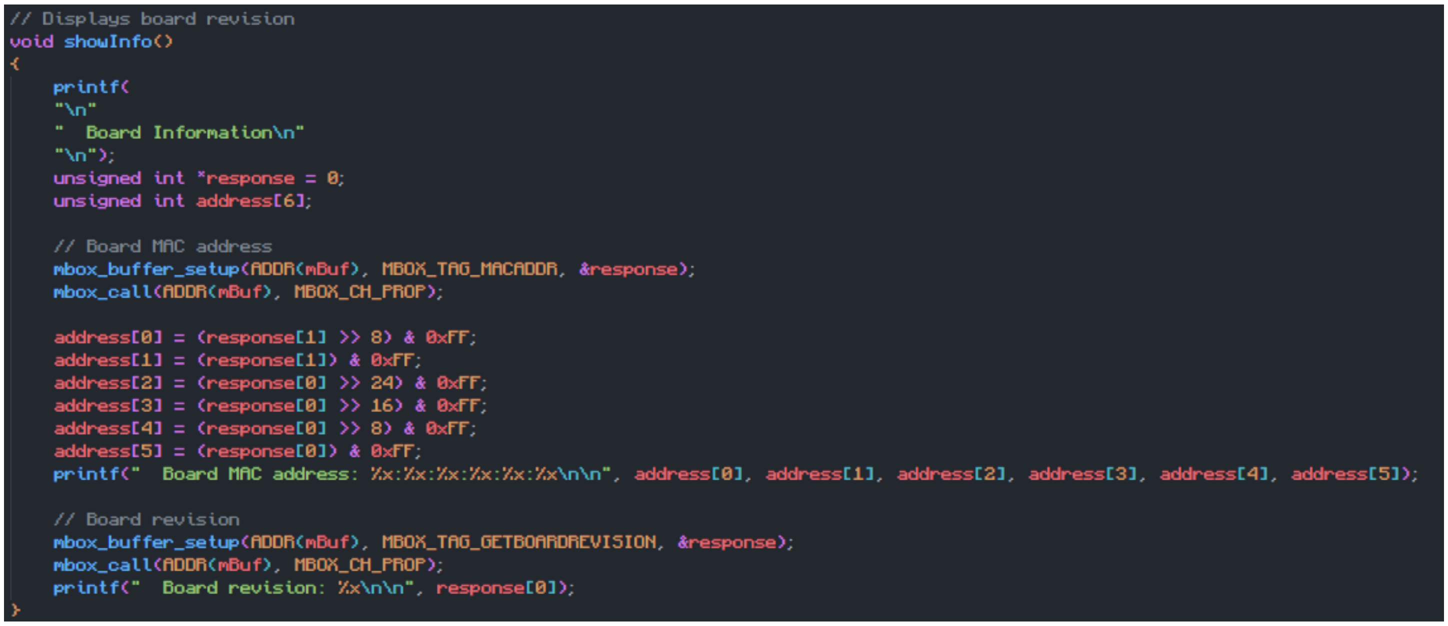
```
Door05> setcolor -t black -b blue

Door05> help

+-------------------------------------------------------------+
| Command List                                        - [ ] x |
+=============================================================+
| Use 'help <command>' to display detailed instructions for a |
| specific command.                                           |
|                                                             |
| Commands:                                                   |
|                                                             |
| home              - Return to home.                         |
| help              - Display this help message.              |
| clear             - Clear the terminal screen.              |
| setcolor          - Set text and background colors.         |
| showinfo          - Display board revision and MAC address. |
|                                                             |
| setbaud           - Set the UART baud rate.                 |
| setdatabits       - Set the UART data bits.                 |
| setstopbits       - Set the UART stop bits.                 |
| setparity         - Set the UART parity.                    |
| setflowcontrol    - Set the UART hardware handshake.        |
|                                                             |
| currentuartsettings - Display current UART settings.        |
+-------------------------------------------------------------+

Door05>
```

Show board informations

The showInfo function is designed to provide users with essential information about the hardware platform, specifically the board's MAC address and its revision number. This information is crucial for both system administrators and developers for tasks ranging from network configuration to software compatibility and debugging.

Implementation

The function operates by interfacing with the system's mailbox, which is a mechanism used to communicate between the CPU and the GPU (which handles various system controls in Raspberry Pi devices). The mailbox interface is accessed via specific function calls that allow the retrieval of system information.

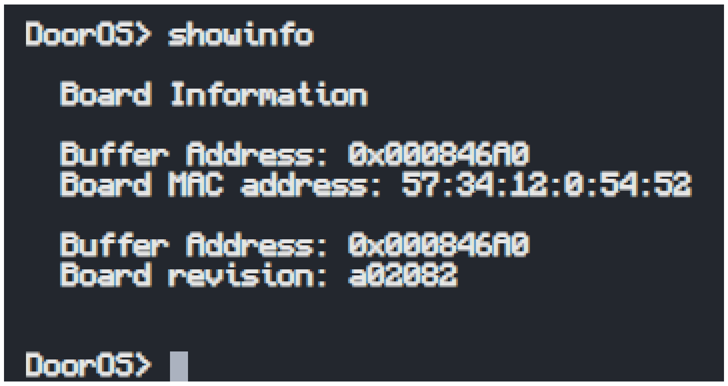


```
// Displays board revision
void showInfo()
{
    printf(
    "\n"
    "  Board Information\n"
    "\n");
    unsigned int *response = 0;
    unsigned int address[6];

    // Board MAC address
    mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_MACADDR, &response);
    mbox_call(ADDR(mBuf), MBOX_CH_PROP);

    address[0] = (response[1] >> 8) & 0xFF;
    address[1] = (response[1]) & 0xFF;
    address[2] = (response[0] >> 24) & 0xFF;
    address[3] = (response[0] >> 16) & 0xFF;
    address[4] = (response[0] >> 8) & 0xFF;
    address[5] = (response[0]) & 0xFF;
    printf("  Board MAC address: %x:%x:%x:%x:%x:%x\n\n", address[0], address[1], address[2], address[3], address[4], address[5]);

    // Board revision
    mbox_buffer_setup(ADDR(mBuf), MBOX_TAG_GETBOARDREVISION, &response);
    mbox_call(ADDR(mBuf), MBOX_CH_PROP);
    printf("  Board revision: %x\n\n", response[0]);
}
```

Mailbox Buffer Setup and function calls:

- **MAC Address:** The MAC address is retrieved by sending a request to the mailbox with the MBOX_TAG_MACADDR tag. The function then reads the response and extracts the MAC address bytes from the returned data. The MAC address is displayed in hexadecimal format.
- **Board Revision:** Similarly, the board revision is retrieved using the MBOX_TAG_GETBOARDREVISION. The response from this request provides the revision number, which is then printed to the console.

Terminal Demonstration:



```
DoorOS> showinfo

  Board Information

  Buffer Address: 0x000846A0
  Board MAC address: 57:34:12:0:54:52

  Buffer Address: 0x000846A0
  Board revision: a02082


DoorOS>
```

The execution of the ***showinfo*** function begins with a call to mbox_buffer_setup, which prepares the buffer for the mailbox communication. This is followed by mbox_call, which sends the buffer to the mailbox and waits for a response. Once the response is received, the function extracts the required information.

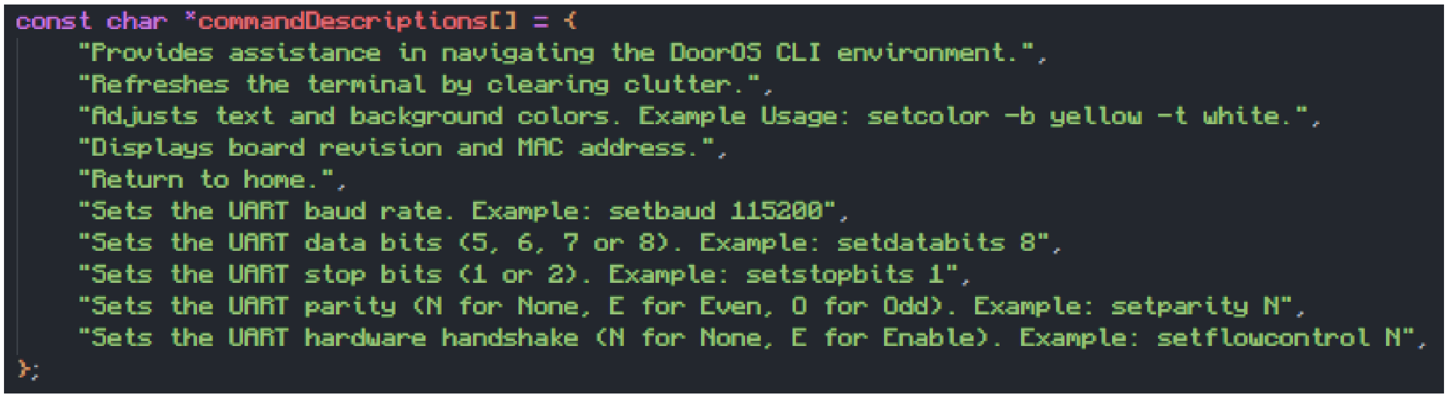
Help and Clear Utility commands

The CLI of DoorOS features several utility commands designed to enhance user interaction and system navigation. Among these, the help and clear commands play pivotal roles in aiding users in command navigation and maintaining a clutter-free terminal environment, respectively.

Help Command Functionality

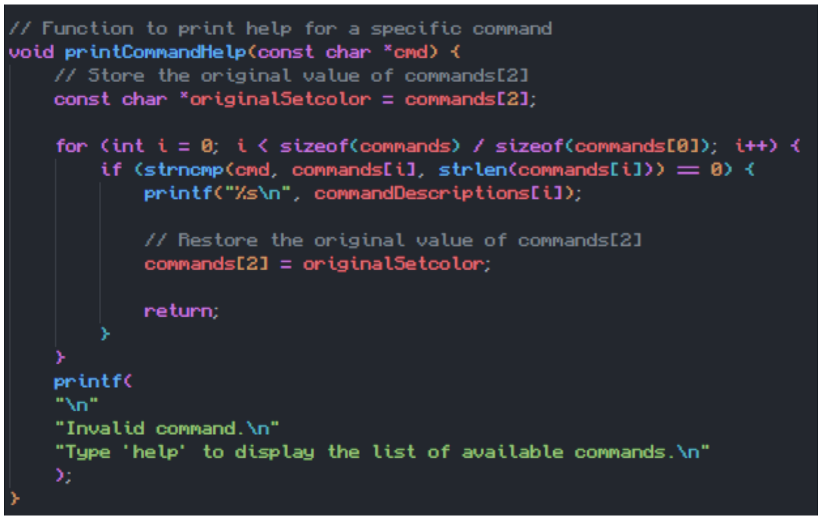
The help command is integral to user support within the CLI. It provides users with information about the available commands and their usage. This feature is particularly useful for both new and experienced users as it ensures they can fully utilize all system functionalities without needing to recall the specifics of each command.

Implementation:



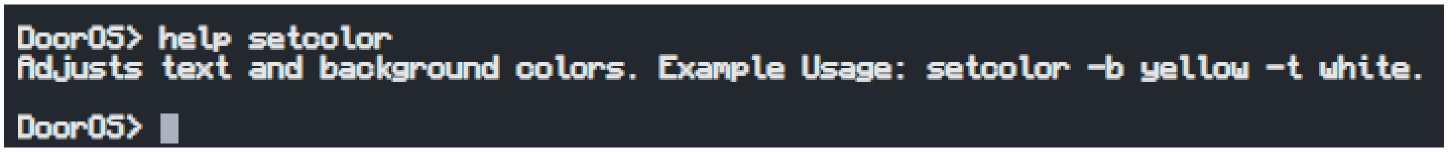
```c
const char *commandDescriptions[] = {
    "Provides assistance in navigating the DoorOS CLI environment.",
    "Refreshes the terminal by clearing clutter.",
    "Adjusts text and background colors. Example Usage: setcolor -b yellow -t white.",
    "Displays board revision and MAC address.",
    "Return to home.",
    "Sets the UART baud rate. Example: setbaud 115200",
    "Sets the UART data bits (5, 6, 7 or 8). Example: setdatabits 8",
    "Sets the UART stop bits (1 or 2). Example: setstopbits 1",
    "Sets the UART parity (N for None, E for Even, O for Odd). Example: setparity N",
    "Sets the UART hardware handshake (N for None, E for Enable). Example: setflowcontrol N",
};
```

- The command looks up the user's input in the command array and displays the associated description from the command descriptions array.

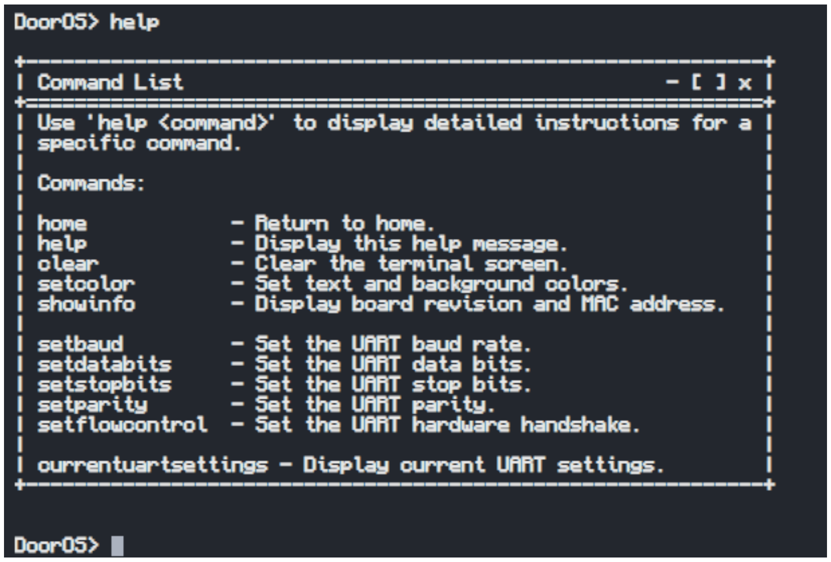


```c
// Function to print help for a specific command
void printCommandHelp(const char *cmd) {
    // Store the original value of commands[2]
    const char *originalSetcolor = commands[2];

    for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
        if (strncmp(cmd, commands[i], strlen(commands[i])) == 0) {
            printf("%s\n", commandDescriptions[i]);

            // Restore the original value of commands[2]
            commands[2] = originalSetcolor;

            return;
        }
    }
    printf(
    "\n"
    "Invalid command.\n"
    "Type 'help' to display the list of available commands.\n"
    );
}
```

- If the input command does not match any known command, it prompts the user with an "Invalid command" message and suggests typing 'help' for a list of available commands.

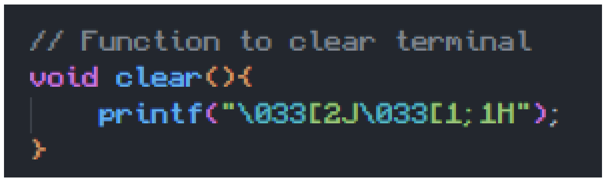


```
DoorOS> help setcolor
Adjusts text and background colors. Example Usage: setcolor -b yellow -t white.

DoorOS> 
```



```
DoorOS> help

+--------------------------------------------------------------+
| Command List                                         - [ ] x |
+==============================================================+
| Use 'help <command>' to display detailed instructions for a |
| specific command.                                            |
|                                                              |
| Commands:                                                    |
|                                                              |
| home             - Return to home.                           |
| help             - Display this help message.                |
| clear            - Clear the terminal screen.                |
| setcolor         - Set text and background colors.           |
| showinfo         - Display board revision and MAC address.   |
|                                                              |
| setbaud          - Set the UART baud rate.                   |
| setdatabits      - Set the UART data bits.                   |
| setstopbits      - Set the UART stop bits.                   |
| setparity        - Set the UART parity.                      |
| setflowcontrol   - Set the UART hardware handshake.          |
|                                                              |
| currentuartsettings - Display current UART settings.         |
+--------------------------------------------------------------+

DoorOS>
```

Clear Command Functionality



```
// Function to clear terminal
void clear(){
    printf("\033[2J\033[1;1H");
}
```

The clear command refreshes the user's terminal window, removing all previous outputs. This command is essential for managing the visual clutter that can accumulate after extensive use of the CLI.

Implementation:

- This command executes an ANSI escape sequence that clears the terminal screen and resets the cursor to the top-left position.
- This simple yet effective command is critical for improving the readability and usability of the terminal, especially during long sessions or when commands generate substantial output.

2. Further Development of UART Driver

The UART (Universal Asynchronous Receiver/Transmitter) driver is a critical component of DoorOS, facilitating communication between the operating system and various peripheral devices through serial ports. This part of the project focuses on extending the capabilities of the existing UART driver to support more advanced configurations and functionalities, enhancing both its flexibility and robustness.

Enhancements to UART Driver

1. **Configurable Baud Rate:**
   - **Objective:** Allow users to set different baud rates according to their specific communication needs.
   - **Implementation:** A command setbaud was introduced, enabling dynamic configuration of the baud rate. Users can specify standard rates such as 9600, 19200, 38400, 57600, and 115200 bps.
2. **Data Bits Configuration:**

- **Objective:** Provide the ability to configure the number of data bits transmitted and received in each UART frame.
- **Implementation:** The setdatabits command allows users to choose between 5, 6, 7, or 8 data bits.

3. **Stop Bits Selection:**
   - **Objective:** Enable selection between one or two stop bits in UART communication.
   - **Implementation:** Introduced the setstopbits command, enhancing the driver's ability to handle different stop bit settings, which is crucial for matching communication specifications of connected devices.
4. **Parity Bit Configuration:**
   - **Objective:** Implement parity bit settings to improve data integrity in communication.
   - **Implementation:** Users can configure the parity as None, Even, or Odd using the setparity command.
5. **Hardware Flow Control:**
   - **Objective:** Manage data flow between devices to prevent loss of data.
   - **Implementation:** The setflowcontrol command allows users to enable or disable RTS/CTS (Request to Send/Clear to Send) hardware flow control.

Technical Implementation

Each of these features was implemented by extending the existing UART driver code with new functions and modifying the UART setup routines to accommodate dynamic changes. This involved:

- Writing functions to update UART control registers based on user inputs.
- Ensuring that the UART driver can safely and effectively manage changes in settings even during runtime.
- Adding comprehensive error checking to prevent incorrect configurations that could lead to system instability or communication failures.

UART Initialization

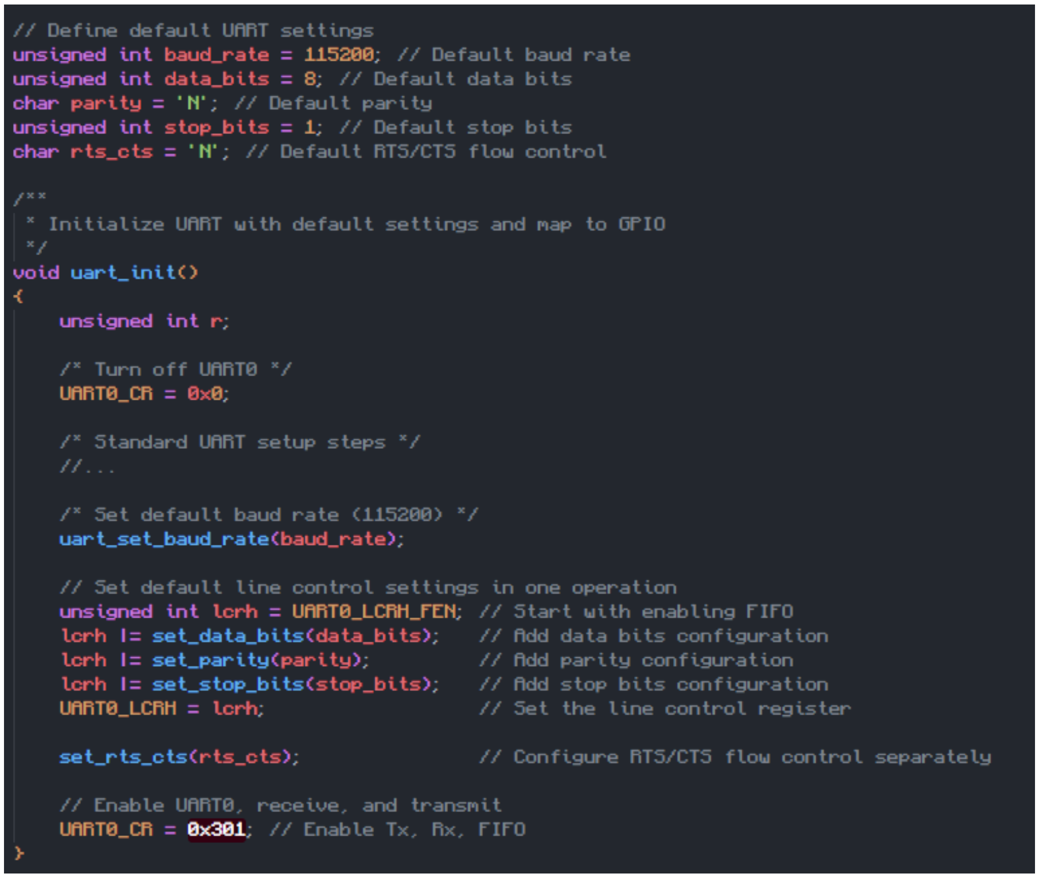
The uart_init function is essential for setting up the UART to ensure it's properly configured for use in DoorOS. This function initializes the UART with default settings, including baud rate, data bits, parity, and stop bits, and maps it to the necessary GPIO pins for communication.

Key steps in the initialization process include:

1. **Disabling the UART** to prevent any data transmission during the setup process.
2. **Configuring the Line Control Register** with default values for data length, stop bits, and optionally, parity bits.

3. **Setting the Baud Rate** to control the speed of data transmission, typically set at a standard 115200 bits per second for reliable high-speed communication.
4. **Enabling FIFO** (First In, First Out) buffers, which help manage incoming data efficiently.
5. **Enabling the UART** for transmission and reception, marking it ready for communication tasks.

This initialization is crucial as it ensures that the UART is configured correctly and ready to handle data transmission and reception reliably within the system.



```
// Define default UART settings
unsigned int baud_rate = 115200; // Default baud rate
unsigned int data_bits = 8; // Default data bits
char parity = 'N'; // Default parity
unsigned int stop_bits = 1; // Default stop bits
char rts_cts = 'N'; // Default RTS/CTS flow control

/**
 * Initialize UART with default settings and map to GPIO
 */
void uart_init()
{
    unsigned int r;

    /* Turn off UART0 */
    UART0_CR = 0x0;

    /* Standard UART setup steps */
    //...

    /* Set default baud rate (115200) */
    uart_set_baud_rate(baud_rate);

    // Set default line control settings in one operation
    unsigned int lcrh = UART0_LCRH_FEN; // Start with enabling FIFO
    lcrh |= set_data_bits(data_bits);   // Add data bits configuration
    lcrh |= set_parity(parity);         // Add parity configuration
    lcrh |= set_stop_bits(stop_bits);   // Add stop bits configuration
    UART0_LCRH = lcrh;                  // Set the line control register

    set_rts_cts(rts_cts);               // Configure RTS/CTS flow control separately

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

Setting the UART Baud Rate

Function Overview

The uart_set_baud_rate function is a key component in the UART driver that allows dynamic adjustment of the baud rate, which is crucial for managing the data transmission speed between the UART and connected devices. The baud rate determines how many bits per second are transmitted, with common settings including 9600, 19200, 38400, 57600, and 115200 bits per second.

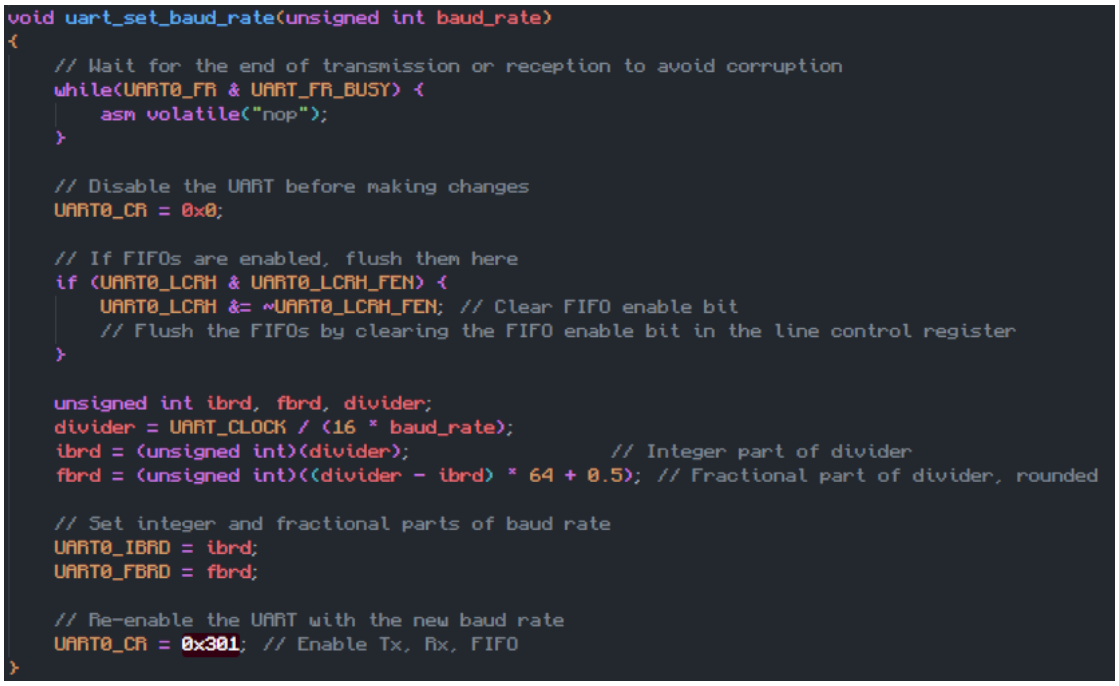
Implementation

Functionality:

1. **Ensure Idle Line:** The function begins by ensuring that there is no ongoing transmission or reception. This is crucial to prevent data corruption during the configuration change.

2. **Disable UART:** Temporarily disables the UART to safely update settings.
3. **Flush FIFOs:** If FIFO (First In, First Out) buffers are enabled, they are flushed to clear any residual data, ensuring a clean start.
4. **Calculate Dividers:** The baud rate is set using a divider calculated based on the UART's clock rate:
    - divider is calculated as the quotient of the UART clock rate divided by 16 times the desired baud rate. This calculation derives from the UART's baud rate setting equation.
    - ibrd (integer part of the divider) and fbrd (fractional part of the divider) are calculated to precisely configure the baud rate.
5. **Set Baud Rate:** The integer and fractional parts of the baud rate divider are set in the UART's IBRD and FBRD registers, respectively.
6. **Re-enable UART:** Finally, the UART is re-enabled with the new baud rate, along with Tx (transmit), Rx (receive), and FIFO settings.

Code Example:



```c
void uart_set_baud_rate(unsigned int baud_rate)
{
    // Wait for the end of transmission or reception to avoid corruption
    while(UART0_FR & UART_FR_BUSY) {
        asm volatile("nop");
    }

    // Disable the UART before making changes
    UART0_CR = 0x0;

    // If FIFOs are enabled, flush them here
    if (UART0_LCRH & UART0_LCRH_FEN) {
        UART0_LCRH &= ~UART0_LCRH_FEN; // Clear FIFO enable bit
        // Flush the FIFOs by clearing the FIFO enable bit in the line control register
    }

    unsigned int ibrd, fbrd, divider;
    divider = UART_CLOCK / (16 * baud_rate);
    ibrd = (unsigned int)(divider);                  // Integer part of divider
    fbrd = (unsigned int)((divider - ibrd) * 64 + 0.5); // Fractional part of divider, rounded

    // Set integer and fractional parts of baud rate
    UART0_IBRD = ibrd;
    UART0_FBRD = fbrd;

    // Re-enable the UART with the new baud rate
    UART0_CR = 0x301; // Enable Tx, Rx, FIFO
}
```

Setting the Number of Data Bits

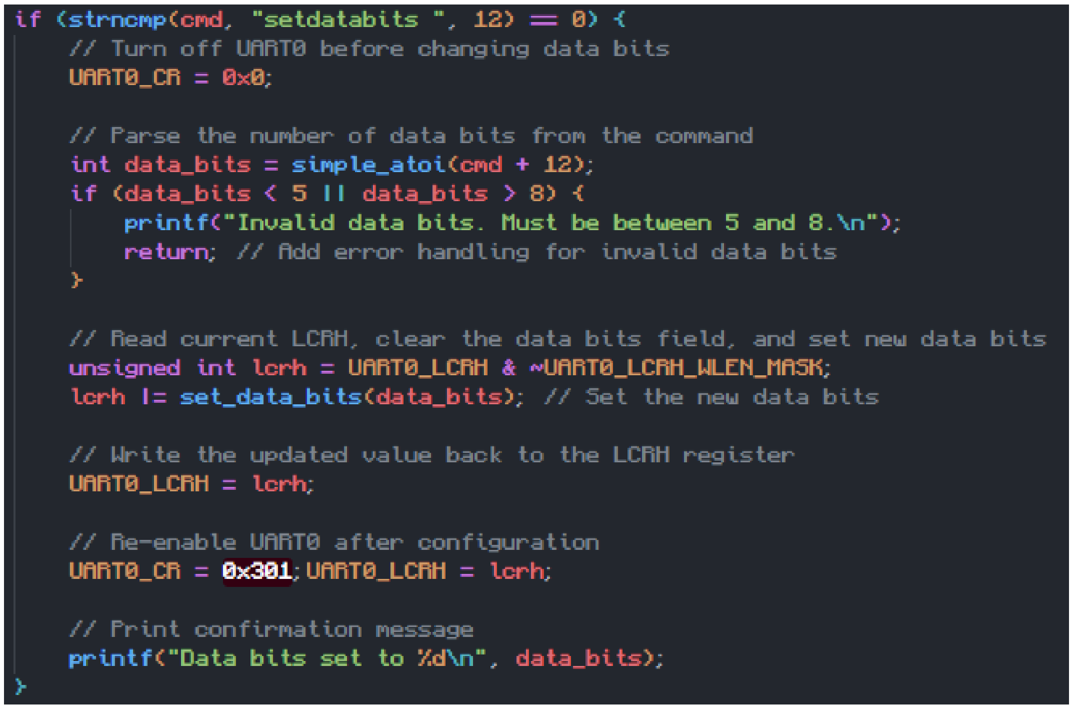
The set_data_bits function allows for the dynamic configuration of the number of data bits in UART frames, which are crucial for defining the structure of the data packets transmitted and received by the UART. Data bits are the actual bits of data being transmitted, excluding the start bit, parity bit, and stop bits.

Implementation

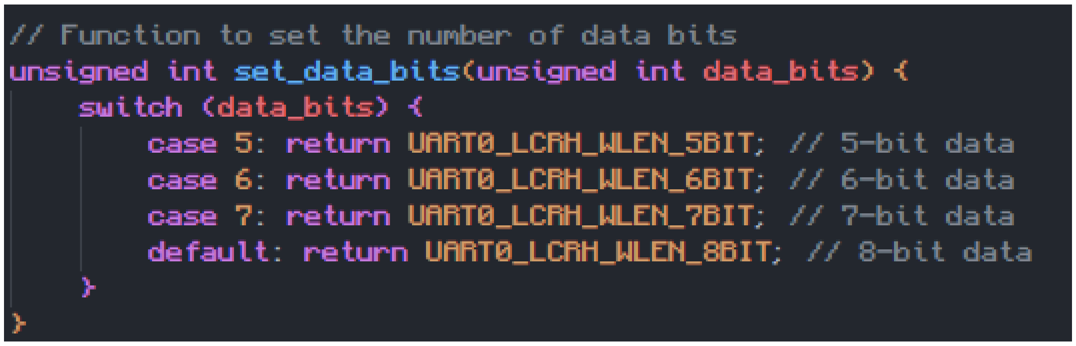
Functionality:

1. **Input Validation:** The function first checks if the provided data_bits value is within the valid range (5 to 8 bits). If not, it returns an error message to the user and exits the function to prevent incorrect configuration.
2. **Disabling UART:** Temporarily disables the UART to safely change settings without data corruption.
3. **Configuring Data Bits:**
   - The function uses a switch-case statement to associate the desired number of data bits with the corresponding setting defined in the UART's Line Control Register High (UART0_LCRH).
   - It modifies only the data bits portion of the LCRH register while preserving other settings (like parity and stop bits settings).
4. **Writing to UART Register:** Updates the LCRH register with the new data bits configuration.
5. **Re-enabling UART:** Reactivates the UART with the new configuration to resume communication operations.

Code Example:



```c
if (strncmp(cmd, "setdatabits ", 12) == 0) {
    // Turn off UART0 before changing data bits
    UART0_CR = 0x0;

    // Parse the number of data bits from the command
    int data_bits = simple_atoi(cmd + 12);
    if (data_bits < 5 || data_bits > 8) {
        printf("Invalid data bits. Must be between 5 and 8.\n");
        return; // Add error handling for invalid data bits
    }

    // Read current LCRH, clear the data bits field, and set new data bits
    unsigned int lcrh = UART0_LCRH & ~UART0_LCRH_WLEN_MASK;
    lcrh |= set_data_bits(data_bits); // Set the new data bits

    // Write the updated value back to the LCRH register
    UART0_LCRH = lcrh;

    // Re-enable UART0 after configuration
    UART0_CR = 0x301; UART0_LCRH = lcrh;

    // Print confirmation message
    printf("Data bits set to %d\n", data_bits);
}
```



```c
// Function to set the number of data bits
unsigned int set_data_bits(unsigned int data_bits) {
    switch (data_bits) {
        case 5: return UART0_LCRH_WLEN_5BIT; // 5-bit data
        case 6: return UART0_LCRH_WLEN_6BIT; // 6-bit data
        case 7: return UART0_LCRH_WLEN_7BIT; // 7-bit data
        default: return UART0_LCRH_WLEN_8BIT; // 8-bit data
    }
}
```

Setting the Number of Stop Bits

Function Overview

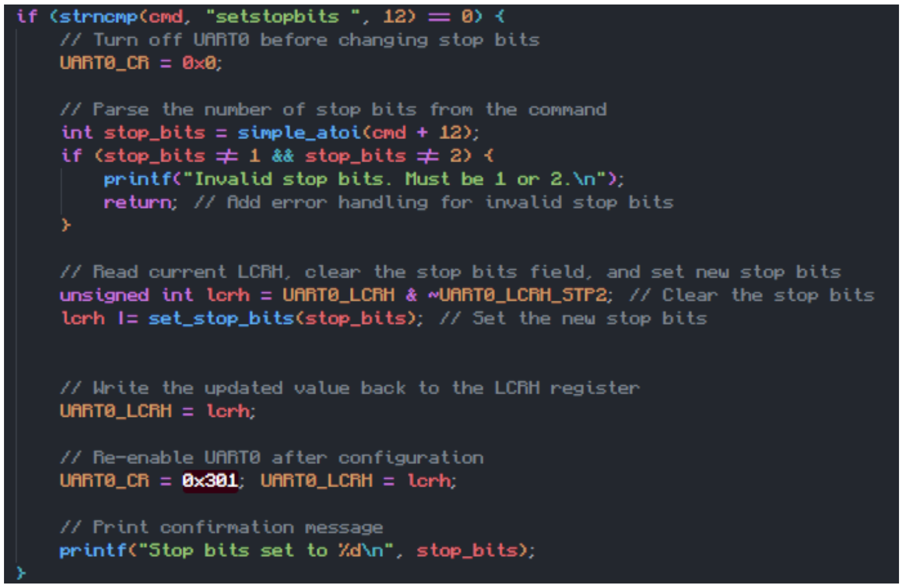
The set_stop_bits function enables dynamic configuration of the number of stop bits in UART frames. Stop bits are used in serial communication to mark the end of a data packet, making them crucial for the correct framing and timing of data reception.

Implementation

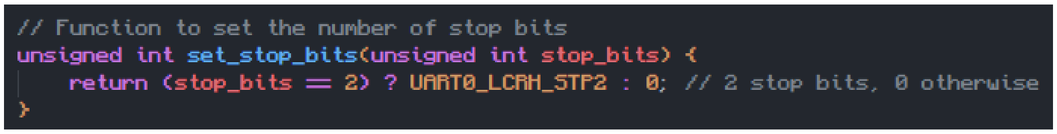
Functionality:

1. **Input Validation:** The function first verifies that the stop_bits input is either 1 or 2. Any other value is considered invalid, and the function will notify the user and terminate the adjustment process to prevent misconfigurations.
2. **Disabling UART:** Temporarily disables the UART to prevent data corruption and ensure changes are safely applied.
3. **Configuring Stop Bits:**
   - The current settings in the Line Control Register High (UART0_LCRH) are read, and the stop bits field is cleared.
   - Based on the desired number of stop bits, the appropriate bit setting is applied using a conditional operation. The function set_stop_bits determines the correct register value to represent the desired stop bits.
4. **Writing to UART Register:** Updates the LCRH register with the new configuration for stop bits.
5. **Re-enabling UART:** Reactivates the UART with the new configuration settings to resume normal operations.

Code Example:



```c
if (strncmp(cmd, "setstopbits ", 12) == 0) {
    // Turn off UART0 before changing stop bits
    UART0_CR = 0x0;

    // Parse the number of stop bits from the command
    int stop_bits = simple_atoi(cmd + 12);
    if (stop_bits ≠ 1 && stop_bits ≠ 2) {
        printf("Invalid stop bits. Must be 1 or 2.\n");
        return; // Add error handling for invalid stop bits
    }

    // Read current LCRH, clear the stop bits field, and set new stop bits
    unsigned int lcrh = UART0_LCRH & ~UART0_LCRH_STP2; // Clear the stop bits
    lcrh |= set_stop_bits(stop_bits); // Set the new stop bits


    // Write the updated value back to the LCRH register
    UART0_LCRH = lcrh;

    // Re-enable UART0 after configuration
    UART0_CR = 0x301; UART0_LCRH = lcrh;

    // Print confirmation message
    printf("Stop bits set to %d\n", stop_bits);
}
```



```c
// Function to set the number of stop bits
unsigned int set_stop_bits(unsigned int stop_bits) {
    return (stop_bits == 2) ? UART0_LCRH_STP2 : 0; // 2 stop bits, 0 otherwise
}
```

Configuring Parity

Function Overview

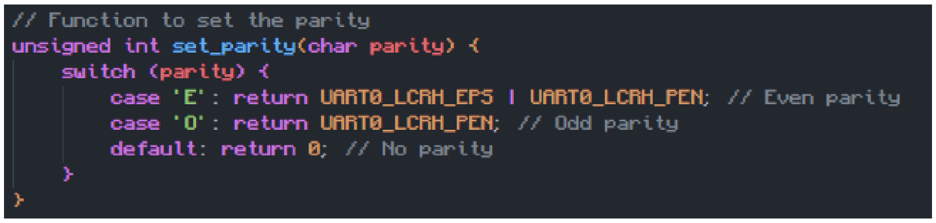
The set_parity function enables users to specify the parity setting for UART communication. Parity is an error-checking feature used to ensure the integrity of the data transmitted, making it crucial for applications where data correctness is vital.

Implementation

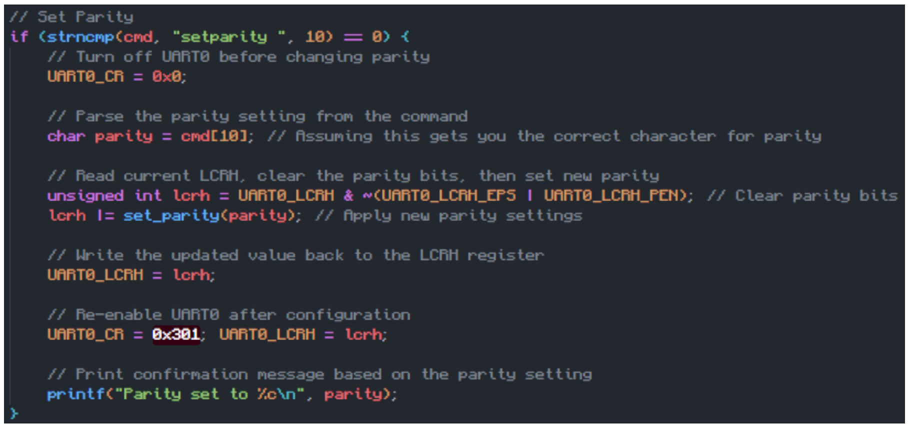
Functionality:

1. **Input Validation and Parsing:**
    - The function starts by parsing the character that represents the desired parity from the command string. This character is then used to determine the appropriate parity setting.
2. **Disabling UART:**
    - Temporarily disables the UART to safely change settings and prevent data corruption during the modification process.
3. **Configuring Parity:**
    - The current settings in the Line Control Register High (UART0_LCRH) are modified by first clearing existing parity settings.
    - Based on the parsed parity character, the appropriate parity configuration is applied using bitwise operations. The function set_parity decides the correct register values to represent the desired parity setting.
4. **Writing to UART Register:**
    - Updates the LCRH register with the new parity configuration.
5. **Re-enabling UART:**
    - Reactivates the UART with the new settings to resume normal operations.

Code Example:



```c
// Function to set the parity
unsigned int set_parity(char parity) {
    switch (parity) {
        case 'E': return UART0_LCRH_EPS | UART0_LCRH_PEN; // Even parity
        case 'O': return UART0_LCRH_PEN; // Odd parity
        default: return 0; // No parity
    }
}
```



```c
// Set Parity
if (strncmp(cmd, "setparity ", 10) == 0) {
    // Turn off UART0 before changing parity
    UART0_CR = 0x0;

    // Parse the parity setting from the command
    char parity = cmd[10]; // Assuming this gets you the correct character for parity

    // Read current LCRH, clear the parity bits, then set new parity
    unsigned int lcrh = UART0_LCRH & ~(UART0_LCRH_EPS | UART0_LCRH_PEN); // Clear parity bits
    lcrh |= set_parity(parity); // Apply new parity settings

    // Write the updated value back to the LCRH register
    UART0_LCRH = lcrh;

    // Re-enable UART0 after configuration
    UART0_CR = 0x301; UART0_LCRH = lcrh;

    // Print confirmation message based on the parity setting
    printf("Parity set to %c\n", parity);
}
```

Configuring Hardware Flow Control

Function Overview

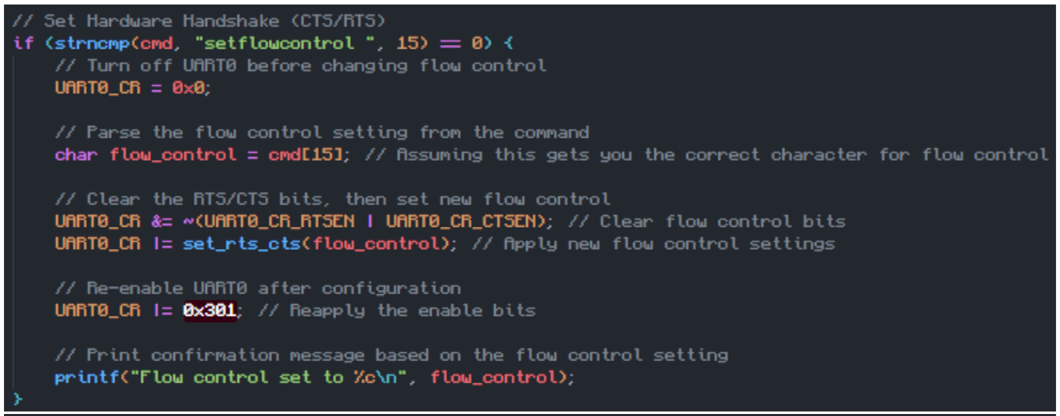
The setflowcontrol function allows users to enable or disable hardware-based flow control, specifically RTS/CTS, in UART communication. Hardware flow control is essential for managing the rate of data transmission between devices, preventing data loss when the receiving device cannot process incoming data as quickly as it is being sent.

Implementation

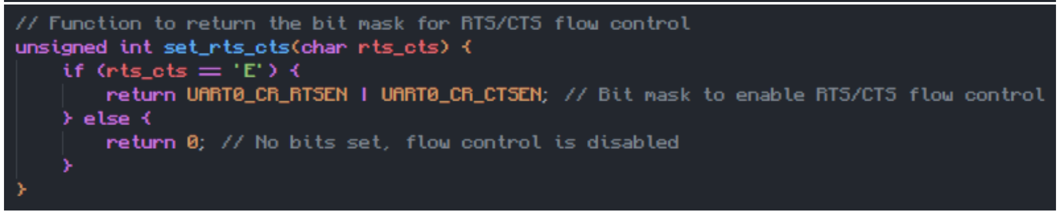
Functionality:

1. **Input Validation and Parsing:**
    - Parses the flow control setting from the command input, which determines whether RTS/CTS flow control should be enabled or disabled.
2. **Disabling UART:**
    - Temporarily disables the UART to allow safe modification of the flow control settings.
3. **Configuring Flow Control:**
    - Clears existing RTS/CTS configuration bits in the UART Control Register (UART0_CR).
    - Based on the parsed flow control character, the appropriate configuration is applied using a bitwise operation. The function set_rts_cts determines the correct register values to represent the desired flow control setting.
4. **Writing to UART Register:**
    - Updates the UART0_CR register with the new flow control configuration.
5. **Re-enabling UART:**
    - Reactivates the UART with the new settings to resume normal operations.

Code Example:

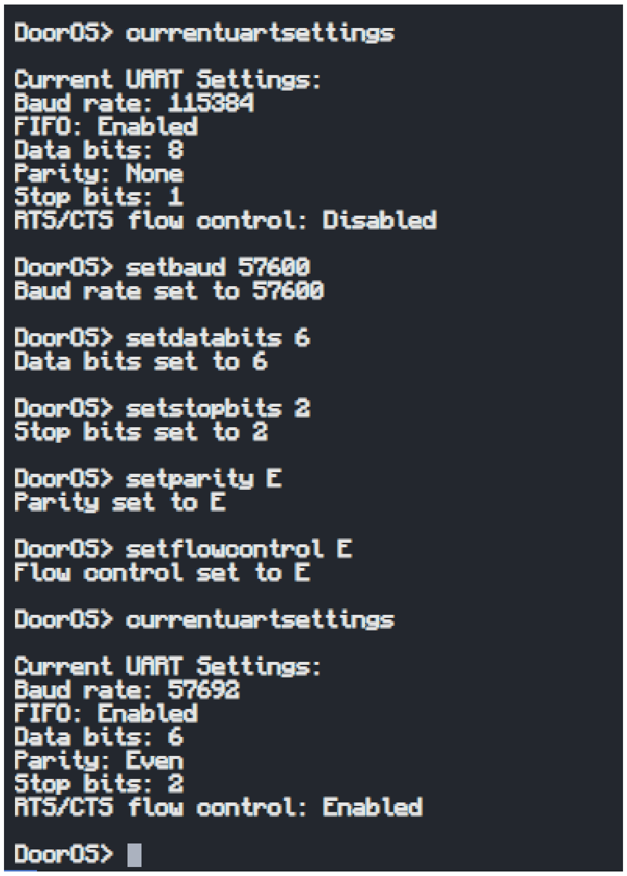


```c
// Set Hardware Handshake (CTS/RTS)
if (strncmp(cmd, "setflowcontrol ", 15) == 0) {
    // Turn off UART0 before changing flow control
    UART0_CR = 0x0;

    // Parse the flow control setting from the command
    char flow_control = cmd[15]; // Assuming this gets you the correct character for flow control

    // Clear the RTS/CTS bits, then set new flow control
    UART0_CR &= ~(UART0_CR_RTSEN | UART0_CR_CTSEN); // Clear flow control bits
    UART0_CR |= set_rts_cts(flow_control); // Apply new flow control settings

    // Re-enable UART0 after configuration
    UART0_CR |= 0x301; // Reapply the enable bits

    // Print confirmation message based on the flow control setting
    printf("Flow control set to %c\n", flow_control);
}
```



```c
// Function to return the bit mask for RTS/CTS flow control
unsigned int set_rts_cts(char rts_cts) {
    if (rts_cts == 'E') {
        return UART0_CR_RTSEN | UART0_CR_CTSEN; // Bit mask to enable RTS/CTS flow control
    } else {
        return 0; // No bits set, flow control is disabled
    }
}
```

Demonstration



```
DoorOS> currentuartsettings

Current UART Settings:
Baud rate: 115384
FIFO: Enabled
Data bits: 8
Parity: None
Stop bits: 1
RTS/CTS flow control: Disabled

DoorOS> setbaud 57600
Baud rate set to 57600

DoorOS> setdatabits 6
Data bits set to 6

DoorOS> setstopbits 2
Stop bits set to 2

DoorOS> setparity E
Parity set to E

DoorOS> setflowcontrol E
Flow control set to E

DoorOS> currentuartsettings

Current UART Settings:
Baud rate: 57692
FIFO: Enabled
Data bits: 6
Parity: Even
Stop bits: 2
RTS/CTS flow control: Enabled

DoorOS> 
```

1. **Initial Settings Display:**
    - Displays the default UART settings including baud rate, data bits, parity, stop bits, and flow control status.
2. **Adjusting UART Settings:**
    - **Baud Rate:** Changed from 115200 to 57600.
    - **Data Bits:** Modified from 8 to 6.
    - **Stop Bits:** Updated from 1 to 2.
    - **Parity:** Set to even (E).
    - **Flow Control:** Enabled RTS/CTS hardware flow control.
3. **Confirmation of Changes:**
    - A final check displays the newly updated settings, confirming that all changes were successfully applied.

Summary of features implemented in both Task 1 & 2

Feature Group		Command/ Feature	Implementation	Testing
Basic Commands		help	Completed	No Issue
		clear	Completed	No Issue
		setcolor	Completed	No Issue
		showinfo	Completed	No Issue
CLI enhancement		OS name in CLI	Completed	No Issue
		Auto-completion in CLI	Completed	No Issue
		Command history in CLI	Completed	No Issue
UART settings	Baud rate	setbaud	Completed	No Issue
	Data bits	setdatabit	Completed	No Issue
	Stop bits	setstopbits	Completed	No Issue
	Parity	setparity	Completed	No Issue
	Hansharking	setflowcontrol	Completed	No Issue

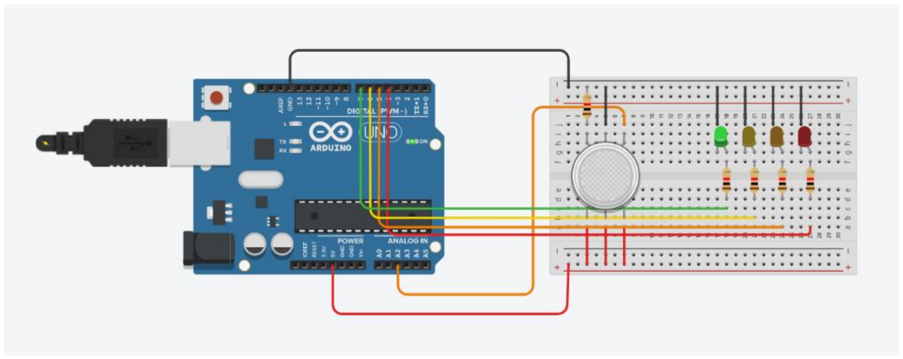
3. Some Common Sensors

Gas Sensors

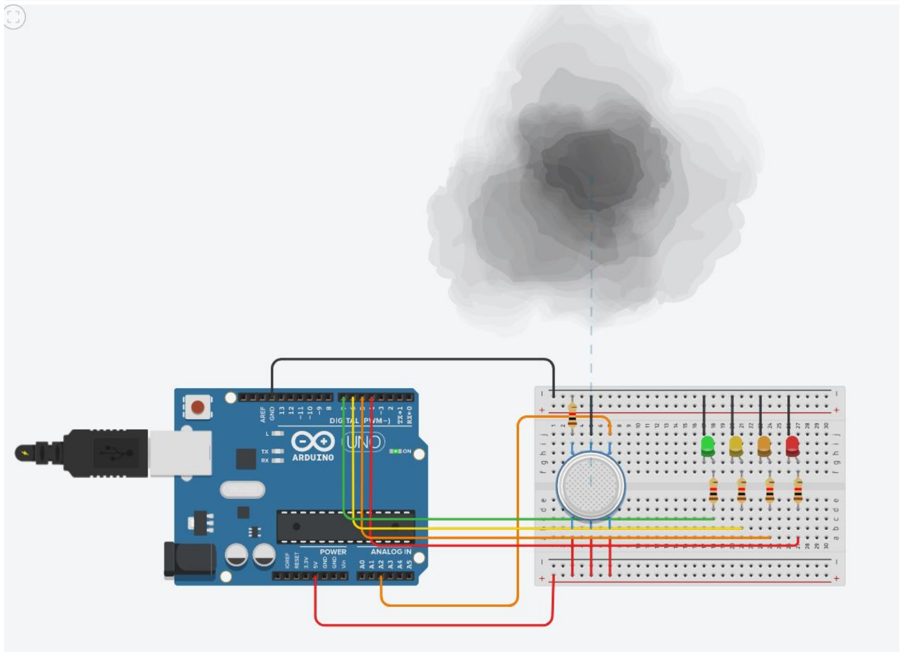
Functionality: Gas sensors are used to detect and measure concentrations of various gases in the environment. They are crucial in safety and environmental monitoring applications.

Working Principle: Most gas sensors operate either through chemical reaction causing changes in resistance or by ionizing the gas and measuring the current flow. The specific working mechanism can vary depending on the type of gas sensor, such as semiconductor, infrared, or electrochemical sensors.

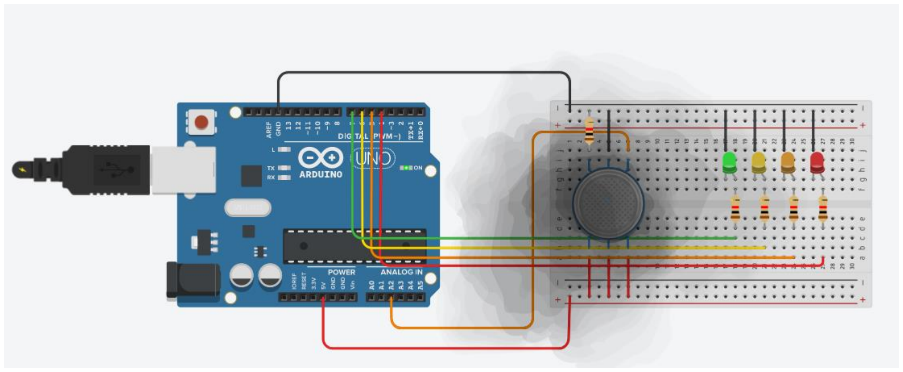
TinkerCAD:



Here I have a simple Gas sensor setup with 4 LED lights representing the severity of the smoke



Here I have simulated a clump of gas, notice the closer the clump of gas is, the more LED gets turns on.



Applications:

- **Industrial Safety:** Monitoring toxic or explosive gas levels to prevent accidents.
- **Environmental Monitoring:** Measuring pollutants in the air, such as carbon monoxide and sulfur dioxide.
- **Home Safety:** Detecting natural gas leaks or carbon monoxide buildup in residential settings.

Market Examples:

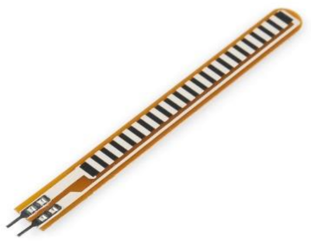
- **Home Safety Detectors:** Products like the Nest Protect use advanced multi-gas sensors capable of detecting carbon monoxide and smoke. These are often more sensitive and faster-reacting compared to traditional binary sensors found in basic smoke alarms.

- **Industrial Safety Monitors:** Devices like Honeywell's GasAlert series offer robust solutions for detecting multiple gases with high precision, often used in hazardous environments.

Comparison:

- **Sensitivity and Range:** Industrial sensors typically offer higher sensitivity and a broader range of detectable gases compared to consumer-grade sensors.
- **Lifespan and Maintenance:** Professional-grade sensors are designed for longer operational life and lower maintenance compared to simpler home sensors.

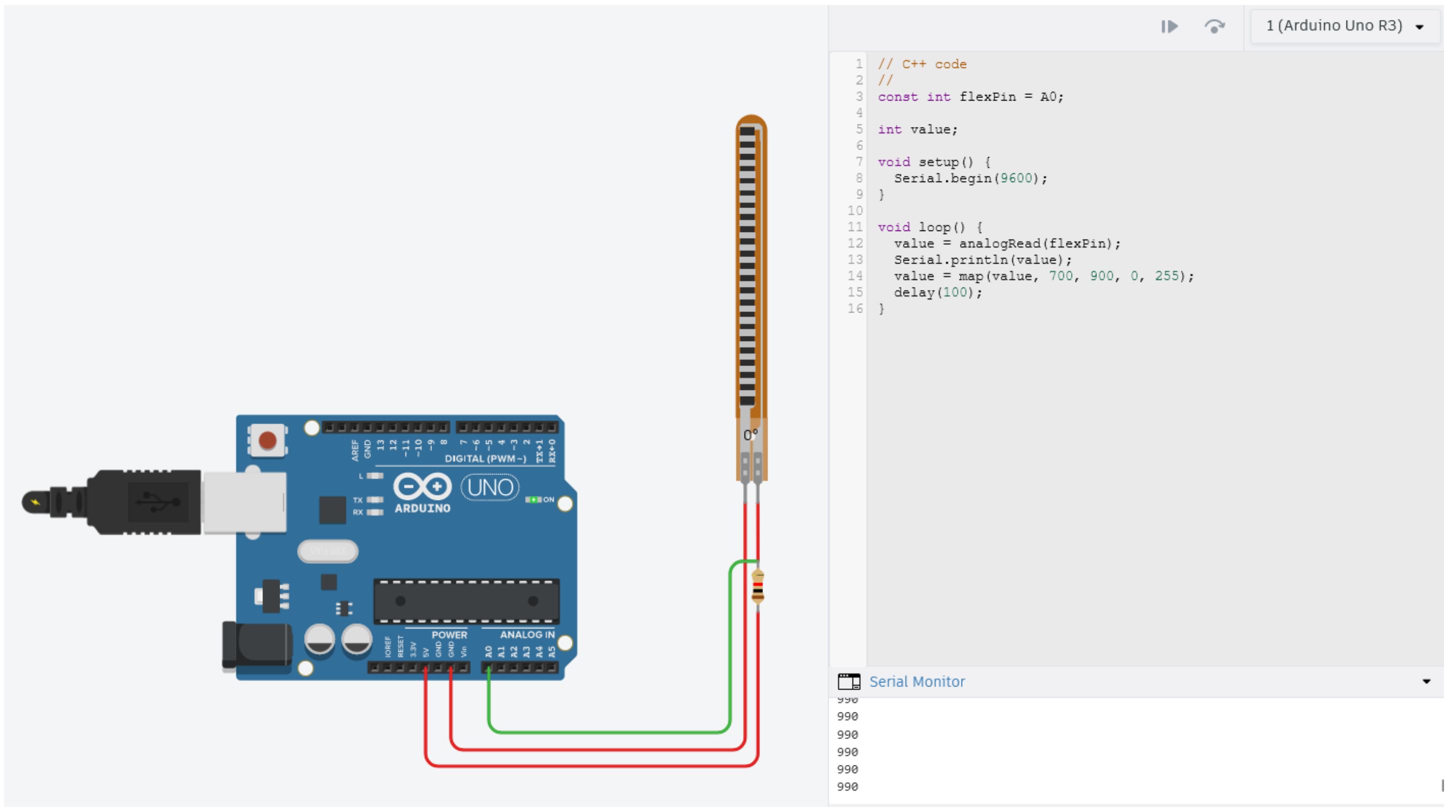
Flex Sensors



Functionality: Flex sensors change in resistance as they are bent. They are used to detect the amount of deflection or bending.

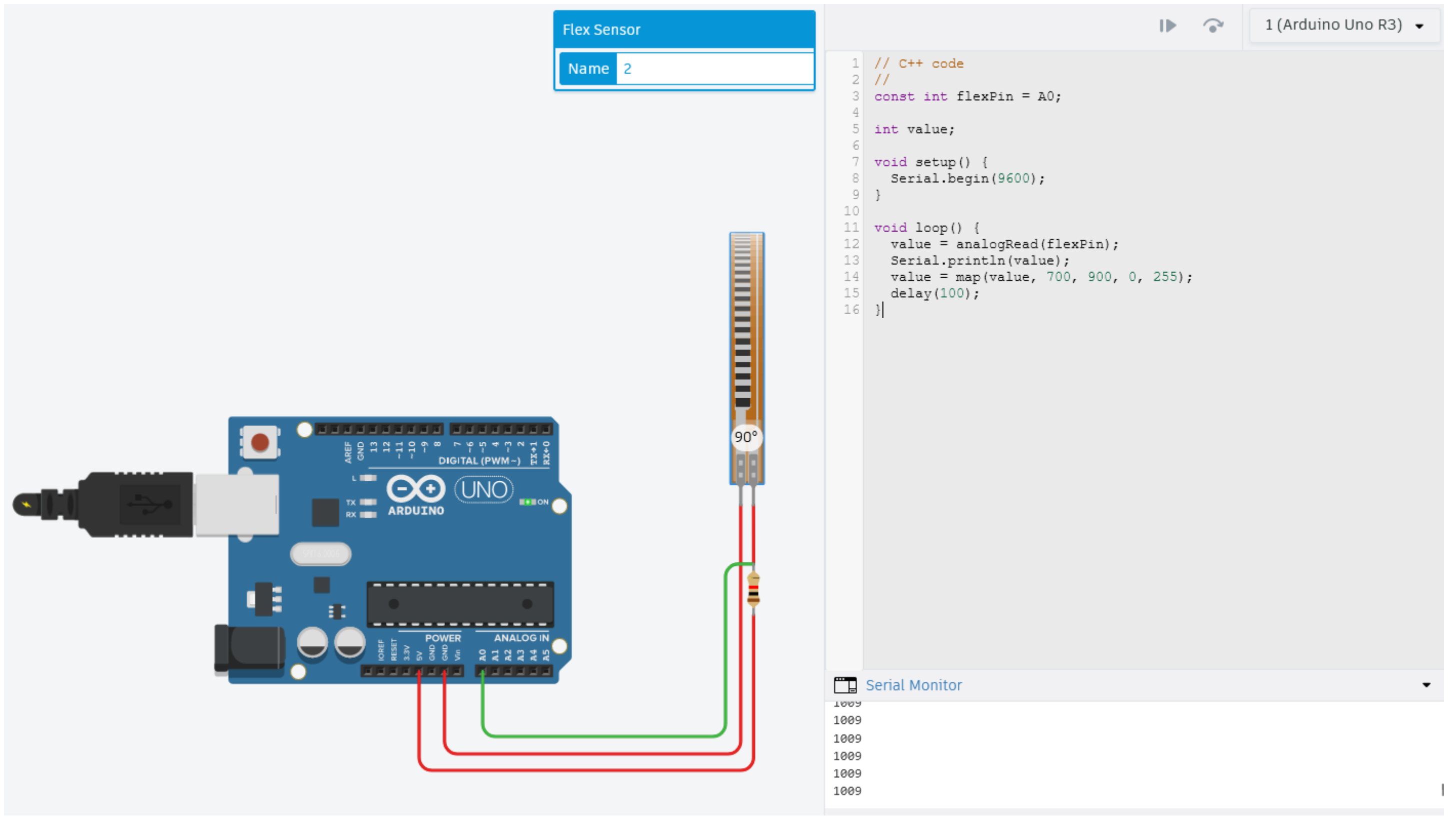
Working Principle: A flex sensor features a conductive polymer that changes resistance as it bends. This change in resistance is measured to determine the angle of bending.

TinkerCAD:



```
// C++ code
//
const int flexPin = A0;

int value;

void setup() {
  Serial.begin(9600);
}

void loop() {
  value = analogRead(flexPin);
  Serial.println(value);
  value = map(value, 700, 900, 0, 255);
  delay(100);
}
```

```
990
990
990
990
990
```



```
// C++ code
//
const int flexPin = A0;

int value;

void setup() {
  Serial.begin(9600);
}

void loop() {
  value = analogRead(flexPin);
  Serial.println(value);
  value = map(value, 700, 900, 0, 255);
  delay(100);
}
```

```
1009
1009
1009
1009
1009
1009
```

Here you can see that a simple setup that I have, when the flex sensor's value changes as I bend the flex sensor from 0 to 90 degrees with the Serial Motor displaying values confirming the changes in angles.

Applications:

- **Wearable Devices:** Used in gloves or fitness bands to track movement and bending of joints.
- **Robotics:** To monitor and control the motion of robotic limbs.
- **Virtual Reality:** Enhancing interactive experiences by providing real-time input based on the user's gestures.

Market Examples:

- **VR Gloves:** Products like the Manus VR Glove use flex sensors to capture detailed hand movements for virtual reality applications.

- **Health Monitoring Devices:** Wearable technology such as rehabilitation gloves incorporate flex sensors to monitor patient progress in physical therapy.

Comparison:

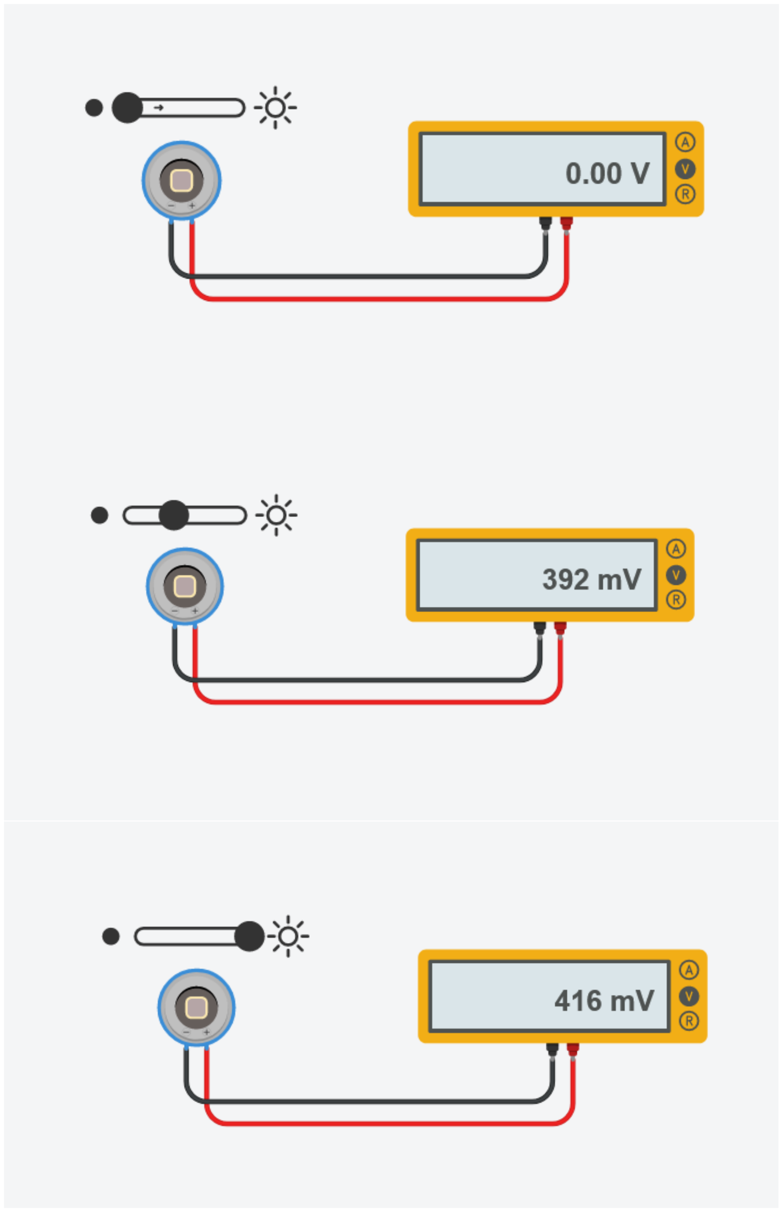
- **Accuracy and Response Time:** VR and professional-grade health monitoring devices usually require higher accuracy and quicker response times than simpler consumer products like toys.
- **Durability:** Sensors in medical or industrial applications are designed for higher durability and repeated use compared to those in consumer electronics.

Photodiodes (Light Sensors)

Functionality: Photodiodes are used to convert light into electrical current or voltage, effectively functioning as light detectors.

Working Principle: A photodiode operates by the photoelectric effect, where light photons hitting the diode create electron-hole pairs that generate a flow of electrical current in a circuit.

TinkerCAD:



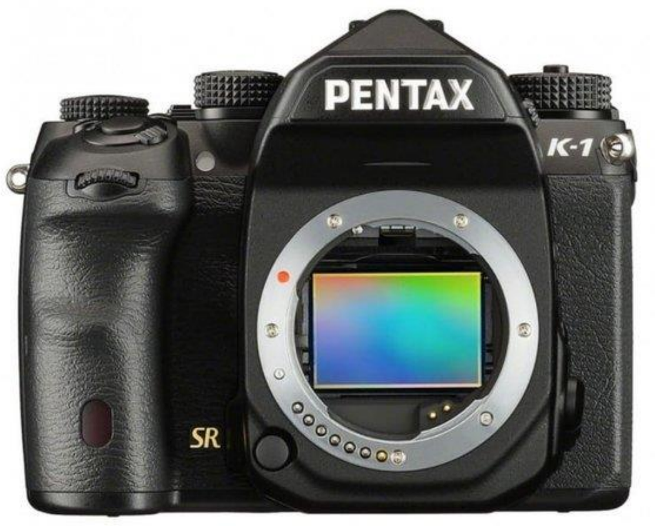
Here I have a simple demonstration, as we can observe, when the photodiode is exposed to more light the voltage in the multimeter increases.

Applications:

- **Optical Communication:** As receivers in fiber optic networks.
- **Consumer Electronics:** In devices such as digital cameras and smartphones to adjust screen brightness based on ambient light.
- **Medical Devices:** In various sensors and instruments for diagnostic purposes such as pulse oximeters.

Market Examples:

- **Digital Cameras:** Photodiodes are used in DSLRs and mirrorless cameras for autofocus functionality by measuring the intensity of light.



- **Ambient Light Sensors in Smartphones:** Devices like the iPhone use ambient light sensors [2] to adjust screen brightness automatically.

Comparison:

- **Sensitivity and Precision:** Photodiodes in professional-grade cameras are typically more sensitive and precise compared to those used in consumer electronics like smartphones.
- **Speed of Response:** Cameras require fast-responding photodiodes to adjust focus quickly in varying light conditions, whereas ambient light sensors in phones and tablets prioritize power efficiency over speed.

III. Reflection & Conclusion

Throughout this project, I successfully implemented a CLI and further enhancement for the UART of DoorOS, my CLI can successfully execute a welcome message upon boot up, perform utility task such as help, setcolor, and clear screen, retrieve board information and most importantly perform UART configuration. This project not only solidified my foundational skills in embedded systems but also expanded my practical understanding of low-level programming and system integration. One of the most daunting challenges I face during the development was Task 2, which involved enhancing the UART driver. Adjusting settings such as baud rate, data bits, and flow control required a deep understanding of both the hardware and its firmware interface. The complexity of these concepts initially overwhelmed me, but through persistent study, consultation with peers, and with the help of Professor Linh I managed to overcome these hurdles, significantly boosting my problem-solving skills.

This project provided a unique opportunity to apply theoretical knowledge in a practical setting, particularly in optimizing system performance and enhancing user interaction through a custom CLI. These experiences are directly applicable to the real-world demands of companies like Razer Inc., where optimizing Linux distributions for gaming platforms is crucial [3].

Throughout this project, I developed critical skills that are highly valued in the embedded software engineering job market. I honed my skills in C/C++ and Python, particularly in writing efficient, low-level code for hardware interfacing, which is a key requirement in job roles at both Razer Inc. and Dyson [4]. I learned to implement and optimize various system settings, such as UART configurations for better performance, aligning with the technical demands seen in the Dyson job description for real-time embedded software development [4].

Reflecting on this project, I appreciate not only the technical skills acquired but also the critical insights gained into the professional applications of embedded systems. Building projects like this from the ground up help me realize that even a simple program with minimal feature where the only user interface is typing request from a keyboard requires a lot of work and effort. This experience has profoundly influenced my interest in low level programming, affirming my passion for embedded software engineering and motivating me to want to learn more about this field.

IV. REFERENCES (USE IEEE STYLES)

[1]P. Loshin and A. S. Gillis, "command-line interface (CLI)," *SearchWindowsServer*, Dec. 03, 2021. [Online]. Available: https://www.techtarget.com/searchwindowsserver/definition/command-line-interface-CLI

[2]"iPhone 14 Pro's New Ambient Light Sensor – Inside the Package | TechInsights," Jan. 25, 2023. [Online]. Available: https://www.techinsights.com/blog/iphone-14-pros-new-ambient-light-sensor-inside-packageiPh

[3]"Razer Inc. hiring Software (Systems Development) Intern in Singapore | LinkedIn." [Online]. Available: https://www.linkedin.com/jobs/view/3806427634/?eBP=CwEAAAGPL44Y-
JzQxgw0o4Sh4zx4vpYlbDC_ejl2XHcyYwyGeCjSQWmGAmYNPN1EBjkuHM6nsiRothwZ3cIolPUoQFwD4D89g4pS
1EBW-yLCRrEPJTHNJ6U-
Jqgur4FX80t_pzRp8EuOjdN6HNKAXeUvtGstydfS69PGalJm2L05B_wTRa3AeSfDWYzB6e_jTUXGhnU2rCFAERQEEG
0pk7QWOYn-MqVq5-
O4bSs5fwu5Gh7crkHR7cNjhbJzghQ0_tGEfjS3CtgMn_P9NgIwu2YqdppPJ5x38cGakiriRcvWq_0k5OnrQlEyT_hj7op
Qv2HkTD6o7ojh21EHpCPEym5-
1IfMwMtPSv1iNvQ&refId=c6%2FiqeajGHi44glVplxIaw%3D%3D&trackingId=MAGktq7LmqvFouxYDjRxvQ%3D
%3D&trk=flagship3_search_srp_jobs

[4]“Dyson hiring Embedded Software Engineer in Singapore, Singapore | LinkedIn.” [Online]. Available: https://www.linkedin.com/jobs/view/3797410027/?eBP=NOT_ELIGIBLE_FOR_CHARGING&refId=c6%2FiqeajGHi44glVplxIaw%3D%3D&trackingId=kZa%2BzTIog4w8iCUg5M0vpQ%3D%3D&trk=flagship3_search_srp_jobs