SYMBOLA

Serial command interface for the ESP32

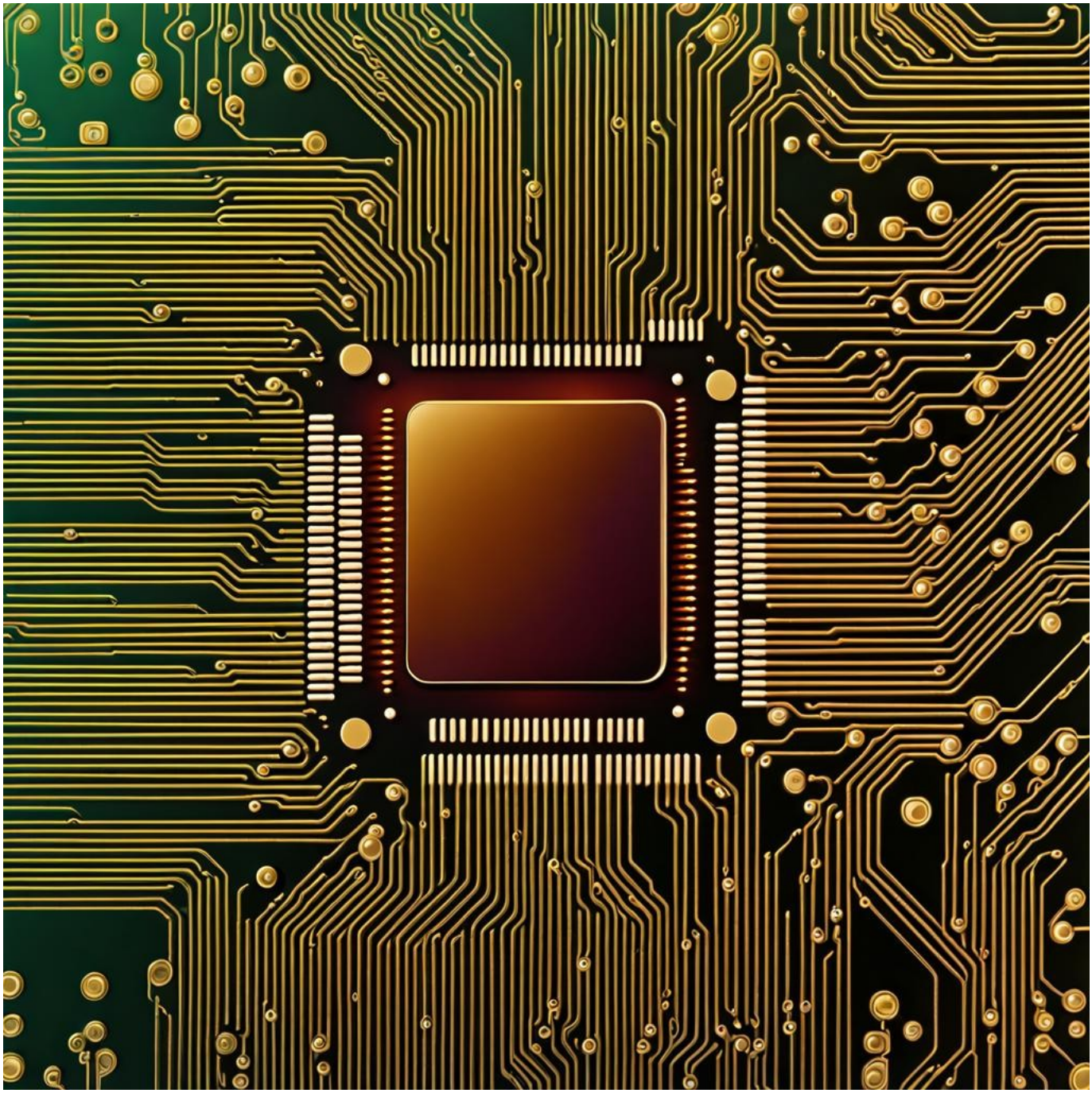

Table of Contents

Disclaimer

This guide is provided for informational purposes only. Every effort has been made to ensure the accuracy and completeness of this guide, but it is provided “as is” without warranty of any kind, express or implied. The author shall not be held liable for any direct, indirect, incidental, or consequential damages or losses arising from the use of this guide.

The procedures and software described in this guide are subject to change and may not be up-to-date. Users are advised to exercise caution and consider their specific circumstances when following the instructions.

This guide may contain links to external websites. The author is not responsible for the content or accuracy of any external site.

Please use this guide responsibly and at your own risk.

Contact the Author

If you've spotted an error or simply wish to make contact, feel free to leave a message at:

https://symbola.co.uk/contact/

Your feedback and inquiries are always welcome!

Preface

The ESP32 is a versatile and popular microcontroller for IoT projects. This guide will show you how to set up a project foundation using FreeRTOS and incorporate a serial command interface. Creating a microcontroller project can be complicated, but this guide aims to simplify the process. It will help you set up FreeRTOS with three demo tasks and establish the serial command interface. This interface is useful for debugging during development, testing, and even for end-users who might find it beneficial. Later in the guide, we will demonstrate how to use the basic interface, add new commands, and effectively utilize FreeRTOS tasks.

If you are new to ESP32 programming within Visual Studio Code using the Arduino framework, take a look at this previous guide for setting up the necessary tools and for tutorials on blinking an LED and printing information through the UART.
https://symbola.co.uk/download/21/esp32/279/sym3.pdf

Hardware / Software Requirements

To follow this guide effectively, you will need the following hardware and software:

1. Windows PC:
   - Operating System: Windows 10 or Windows 11. (Note: This guide uses Windows 11 Pro 64-bit for demonstrations).
2. ESP32 Development Board:
   - Model: ESP32 with 38 Pins. A commonly used model that can be found on AliExpress.
   - https://www.aliexpress.com/item/32959541446.html (ESP-32 38Pin).
3. Micro USB Cable:
   - Used for connecting the ESP32 board to your computer for programming and power supply.
4. LED:
   - A standard light-emitting diode for use in the "Hello World" blinking example.
5. Resistor:
   - A 150-200 Ohm resistor to limit current to the LED, preventing potential damage.
6. Breadboard:
   - A solderless breadboard for assembling the circuit without soldering.
7. Jumper Wires:
   - Also known as linker wires, these are used for making connections on the breadboard between the ESP32, LED, resistor, and other components.

While this guide specifically uses Windows 10/11 for demonstrations, it's worth noting that the ESP32 can also be programmed using other operating systems like macOS and Linux. However, the steps for setting up the development environment and programming the ESP32 may differ in these systems and are not covered in this guide.

Create the PlatformIO project

1. Open Visual Studio Code.
2. Open the PlatformIO extension.
3. Navigate to: Quick Access ► PIO Home ► Open.
4. Create a new project:
   - Name: **UartCLI**
   - Board: **Espressif ESP32 Dev Module**
   - Framework: **Arduino**
   - Location: **Default**
5. Click “Finish” to create the project.
6. Download the source code:
   - Visit: https://github.com/SYMBOLA2024/SYM4
   - Click on the green "Code" button and select “Download ZIP”.
7. Extract the ZIP file.
8. Copy the contents of the “src” and “include” folders from the ZIP to the corresponding folders in your project.
9. There is already a file named “main.cpp” in the destination, replace it with the file from the ZIP.
10. In Visual Studio Code, verify that the code builds by clicking the checkmark at the bottom of the window labeled “PlatformIO: Build”.
11. Connect the ESP32 to your computer and upload the program using the arrow icon labeled “PlatformIO: Upload”.
12. Connect to the ESP32 via its COM port using a terminal program (e.g., MobaXterm) with the following settings:
    - Baud: 115200
    - Data bits: 8
    - Stop bits: 1
    - Parity: None
    - Flow control: Xon/Xoff
13. Upon successful connection, you should see the following messages:
    Starting FreeRTOS tasks...
    Welcome to the ESP32 Command Interface!
    Type 'help' for assistance or 'commands' to list all commands.
    >

Testing the command interface

Try entering the following commands (commands are shown in **red**): -

```
> help
Help will be added soon

> commands
Available commands:
  help
  commands
  test1
  test2
  test3

> test1 "Hello World!"
Executing testFunction1
Arg 1: Hello World!

> test1 A B C D E F
Executing testFunction1
Arg 1: A
Arg 2: B
Arg 3: C
Arg 4: D
Arg 5: E
Arg 6: F

> test1 "A B" "C D" "E F"
Executing testFunction1
Arg 1: A B
Arg 2: C D
Arg 3: E F
```

Adding a new command

To illustrate how to add a new command, this section describes the process of integrating a command to control an LED. This command will have the capabilities to turn the LED on, off, and toggle its state. The LED will be connected to pin 2.

Please add the code highlighted in **bold and red**.
Modify the code indicated in **bold and blue**.

main.h

```
// Constants.
const int SERIAL_BAUD_RATE = 115200;

const int COMMAND_INTERFACE_TASK_STACK_SIZE = 4096;
const int TEST_TASK_STACK_SIZE = 1024;
const int COMMAND_INTERFACE_TASK_PRIORITY = 1;
const int TEST_TASK_PRIORITY_START = 2;
```

main.cpp

```
void setup() {
    // Initialize Serial port for CLI communication & debugging.
    Serial.begin(SERIAL_BAUD_RATE);

    // Wait for the Serial port to connect.
    while (!Serial) {
        delay(10);
    }

    // Initialize the digital pin as an output.
    pinMode(2, OUTPUT);

    // Notify about the start of the program.
    Serial.println("Starting FreeRTOS tasks...");

    // Create tasks.
    createTasks();

    // Start the scheduler.
    vTaskStartScheduler();
}
```

CommandInterfaceTask.h

```
// Declaration of functions to be used in the command interface.
void displayHelp();
void displayCommands();
void ledControl(String args[], int argCount);
void testFunction1(String args[], int argCount);
void testFunction2(String args[], int argCount);
void testFunction3(String args[], int argCount);
```

CommandInterfaceTask.cpp

```
          // Execute the parsed command if valid.
          if (validCommand && argc > 0) {
            // Compare the command against known commands and execute the corresponding function.
            if (argv[0] == "help") {
              displayHelp();
            } else if (argv[0] == "commands") {
              displayCommands();
            } else if (argv[0] == "led") {
              ledControl(argv, argc);
            } else if (argv[0] == "test1") {
              testFunction1(argv, argc);
            } else if (argv[0] == "test2") {
              testFunction2(argv, argc);
            } else if (argv[0] == "test3") {
              testFunction3(argv, argc);
            }
          }
```

```
// Function to display available commands.
void displayCommands() {
  Serial.println("Available commands:");
  Serial.println("  help");
  Serial.println("  commands");
  Serial.println("  led (on/off/toggle)");
  Serial.println("  test1");
  Serial.println("  test2");
  Serial.println("  test3");
}
```

```
// Function to display available commands.
void displayCommands() {
  Serial.println("Available commands:");
  Serial.println("  help");
  Serial.println("  commands");
  Serial.println("  led (on/off/toggle)");
  Serial.println("  test1");
  Serial.println("  test2");
  Serial.println("  test3");
}

void ledControl(String args[], int argCount) {
    if (argCount != 2) { // Command + 1 argument expected
        Serial.println("Error: Incorrect number of arguments. Usage: led <on|off|toggle>");
        return;
    }
    String command = args[1]; // args[1] is the command argument
    command.toLowerCase(); // Convert the command to lowercase
    if (command == "on") {
        digitalWrite(2, HIGH);
        Serial.println("LED turned on");
    } else if (command == "off") {
        digitalWrite(2, LOW);
        Serial.println("LED turned off");
    } else if (command == "toggle") {
        int currentState = digitalRead(2);
        digitalWrite(2, currentState == HIGH ? LOW : HIGH); // Toggle the state
        Serial.println("LED state toggled");
    } else {
        Serial.println("Error: Invalid argument. Use 'on', 'off', or 'toggle'.");
    }
}

// Test function implementations.
void testFunction1(String args[], int argCount) {
  Serial.println("Executing testFunction1");
  for (int i = 1; i < argCount; i++) {
    Serial.print("Arg ");
    Serial.print(i);
    Serial.print(": ");
    Serial.println(args[i]);
  }
}
```

Testing the tasks

To test the additional FreeRTOS tasks (TestTask1, TestTask2, and TestTask3), uncomment the line that enables printing over the serial port. These tasks can be renamed or new ones added as needed.

```
void TestTask1(void *pvParameters) {
  (void) pvParameters; // This line prevents unused variable warning.

  while (1) {
    // Here, you can implement what you want this task to do.
    Serial.println("Test Task 1 is running");

    // Delay for a period of time (e.g., 1 second).
    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}
```

If uncommented in TestTask1, TestTask2, and TestTask3, the message 'Test Task 1 is running' will print every one second, 'Test Task 2 is running' will print every two seconds, and 'Test Task 3 is running' will print every three seconds. These intervals can be modified by adjusting the 'vTaskDelay' duration, as highlighted above, in milliseconds (ms).

Other factors to consider are the amount of memory assigned to each task (stack size) and the priority of the task.

```
    // Create Test Task 1.
    if (xTaskCreate(
            TestTask1,
            "TestTask1",
            TEST_TASK_STACK_SIZE,
            NULL,
            TEST_TASK_PRIORITY_START,
            NULL
        ) != pdPASS) {
        Serial.println("Failed to create Test Task 1");
        return;
    }
```

Final thoughts

With the information provided, you should now have the capability to add new commands for operation via the serial port, as well as integrate additional FreeRTOS tasks. This process enables the customization and expansion of your project, allowing for more sophisticated control and task management. By leveraging these skills, you can significantly enhance the functionality of your device, catering to a wider range of applications and use cases.