Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 1.1

Discipline: Embedded Systems

Elaborated: Berco Andrei, FAF - 221

Checked: asist. univ., Martiniuc A.

Chișinău 2025

Analysis of the Situation in the Field

1. Description of the Technologies Used and Application Context

In this laboratory work, we focus on serial communication and user interaction with a microcontroller using the STDIO library. This allows data exchange between the microcontroller and a computer through a serial port (UART).
This method is frequently used in embedded systems for:

- **Diagnostics and debugging** (displaying status messages).
- **Communication between devices** (e.g., microcontroller - PC).
- **Peripheral control** (e.g., turning an LED on or off, controlling motors, sensors).

In this application, the user can control an LED using text commands sent through a serial terminal. The commands are interpreted and executed, providing a response through the same serial interface.

2. Overview of the Hardware and Software Components Used

Hardware Components

1. **Development Board:**
    - We use the **Arduino Mega 1280**, which provides multiple UART ports and sufficient hardware resources for the application.
2. **LED + 220Ω Resistor:**
    - The LED is controlled through a digital pin, and the resistor limits the current.
3. **Breadboard Connections + Jumper Wires:**
    - Ensure flexible and easy-to-modify circuit connections.
4. **USB Port and UART Converter:**
    - Used for communication between the PC and the microcontroller.

Software Components

1. **PlatformIO (in Visual Studio Code):**

- Used for writing and compiling C/C++ code for Arduino.

2. **STDIO Library:**
   - Allows the use of printf() and scanf() functions for handling serial input and output.
3. **Serial Monitor (Putty / PlatformIO Serial Monitor):**
   - Enables sending and receiving serial commands.
4. **Proteus (optional, for simulation):**
   - If hardware is unavailable, we can test the circuit virtually.

3. System Architecture Explanation and Solution Justification

The application is built on a modular architecture with the following components:

- **serial_comm module** – Manages serial interaction.
- **led_control module** – Controls the LED based on received commands.
- **main.cpp** – The starting point of the application, where the modules are initialized.

Why this solution?

- **Separation of Responsibilities:** Each module handles a single functionality.
- **Reusability:** The code can be extended and reused in other projects.

4. Case Study: LED Control in an Industrial Automation System

Context and Necessity

In the field of industrial automation, serial communication is essential for interaction between equipment, sensors, and operators. A system for monitoring and controlling industrial equipment needs to be fast, reliable, and easy to operate. An operator should be able to quickly check the status of a machine and intervene when necessary without being physically present.

In this context, a microcontroller-based system using a serial interface for sending commands represents an efficient and affordable solution. This case study examines the use of an **Arduino Mega 2560** microcontroller for controlling signal LEDs on a production line.

Practical Implementation

In a production factory, each assembly line is equipped with an **Arduino Mega 2560** microcontroller connected to a central computer. Operators in the control room can use text commands sent via the serial port to activate or deactivate the signal LEDs of the machines.

Each LED has a well-defined role in the system:

- **Green LED** – Indicates the machine is operating normally.
- **Red LED** – Signals an error in machine operation.
- **Blue LED** – Indicates the machine is in standby mode.

Examples of Commands:

1. "led green on" – Activates the green LED, signaling normal operation.
2. "led red on" – Turns on the red LED, indicating a malfunction.
3. "led blue off" – Turns off the blue LED, signaling the end of standby mode.

Workflow in Such a System:

1. A sensor detects an issue with the machine's operation.
2. The system sends a serial message to the microcontroller, such as "led red on".
3. The red LED lights up on the control panel, signaling an error to the operator.
4. After resolving the issue, the operator sends the command "led red off", and the red LED turns off.

Extending the Case Study

This solution can be adapted for other industrial applications, such as:

- Automating conveyor belts, allowing operators to start and stop product transport through serial commands.

- Monitoring temperature and humidity in a warehouse, displaying values on an LCD connected to the microcontroller.
- Controlling robotic arms in assembly lines, where serial commands determine their movement.

Benefits and Impact

Using a control system based on a serial interface offers multiple advantages:

- **Quick Interaction:** Operators can control and monitor equipment remotely.
- **Reliability:** The system is stable and can operate in challenging industrial environments.
- **Scalability:** The solution can be extended to control multiple devices, not just LEDs but also motors, sensors, and other industrial components.
- **Low Cost:** Implementing such a system only requires a microcontroller and monitoring software, eliminating the need for expensive industrial equipment.

Design

1. Architectural Sketch and Component Interconnection

The system architecture is based on the interaction between:

- An **Arduino Mega 2560** microcontroller,
- A **PC** for sending serial commands,
- An **LED** connected to a digital pin of the microcontroller.

Component Description and Their Roles:

- **Arduino Mega 1280:** The main control unit that receives commands via the serial interface and controls the LED.
- **Serial Interface (UART):** Enables communication between the PC and the microcontroller, allowing the user to send commands and receive feedback.

- **LED and 220Ω Resistor:** The LED indicates the response to commands, and the resistor prevents circuit damage.
- **Serial Monitor:** Software on the PC (Putty, PlatformIO Serial Monitor, TeraTerm) used for sending commands and viewing responses.

The architectural sketch illustrates how the main system components interact to enable LED control through the serial interface. Communication between the user and the system is done through a PC equipped with a serial terminal, which sends commands to an **Arduino Mega 2560** microcontroller via the USB/UART interface.

The microcontroller receives the commands and processes them through a dedicated software module. Based on the received instructions, it activates or deactivates the LED using the **D2** pin. The LED is connected to the microcontroller in series with a **220Ω resistor**, which protects the circuit and limits the current through the LED.

To complete the electrical circuit, the cathode of the LED is connected to the **GND** of the microcontroller. The diagram reflects the complete flow of commands, starting from user input, processing by the microcontroller, and physical activation of the LED.

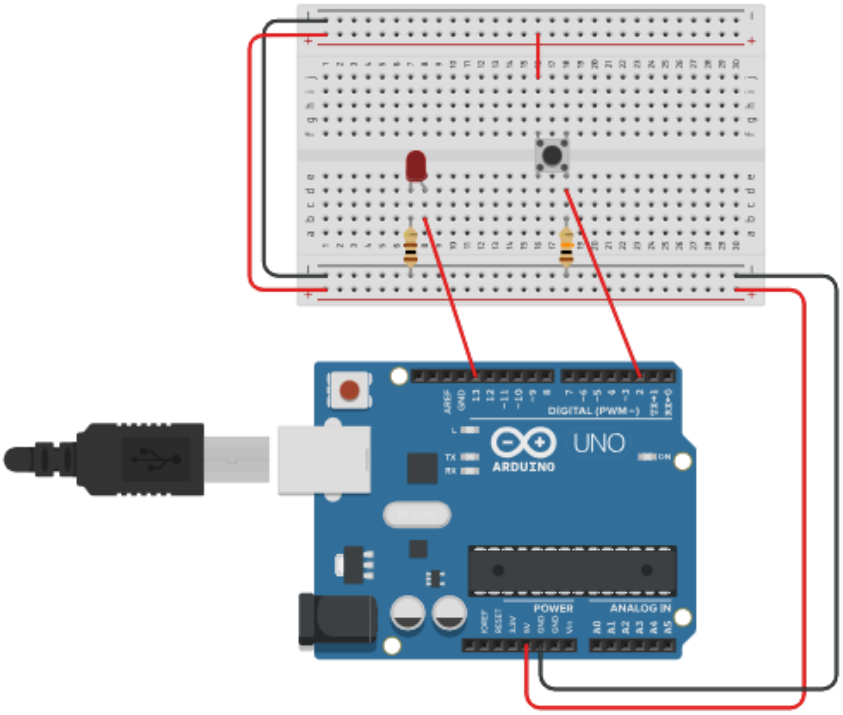

Figure 2.1 Electrical schematic

The electrical schematic shows the detailed connections between the Arduino Mega 2560, LED, 220Ω resistor, and GND. The components are connected in such a way as to allow the LED to be turned on and off through a digital signal generated by the microcontroller.

The control signal is emitted from pin D2 of the Arduino Mega 2560 and is used to turn the LED on or off. This pin is connected to the anode of the LED, while the cathode of the LED is connected to GND through a 220Ω resistor, which serves to limit the current, preventing damage to the LED.

When the user sends a valid command via the serial interface, the microcontroller changes the state of pin D2, thus enabling control of the LED. If the command sent is "led on," the microcontroller sets the pin to HIGH, allowing current to flow and turning on the LED. In the case of the "led off" command, the pin is set to LOW, cutting off the current and turning off the LED.

This schematic clearly illustrates the flow of electric current and the circuit structure, providing a complete representation of the hardware components' interconnection.

2. Scheme bloc și algoritm

To understand the system's behavior, a Flowchart and a Finite State Machine (FSM) are used.

Flowchart – Serial Command Processing
(A Flowchart diagram illustrating the cycle: command reception → processing → execution → user feedback)

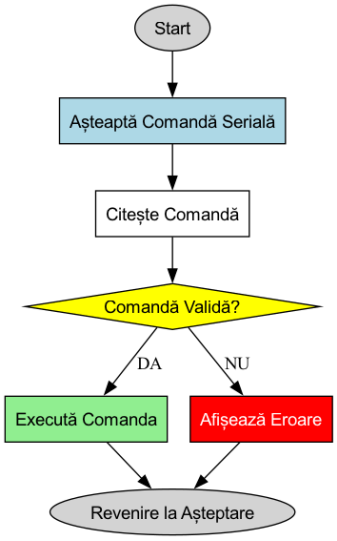

Figure 2.2 Flowchart

The FSM includes the following states:

1. **Idle** – Waits for commands from the user.
2. **Processing** – Processes the received command.
3. **LED ON** – Turns on the LED and confirms via Serial.
4. **LED OFF** – Turns off the LED and confirms via Serial.

5. **Error** – Displays an error message if an invalid command is entered.

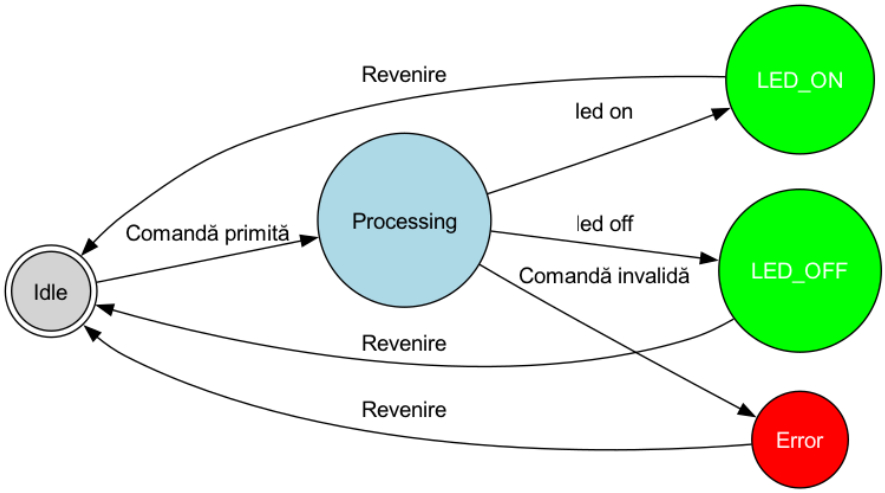

Figure 2.3 FSM diagram

3. Modular implementation

For better project organization, a modular architecture was used, dividing functionalities into separate files.

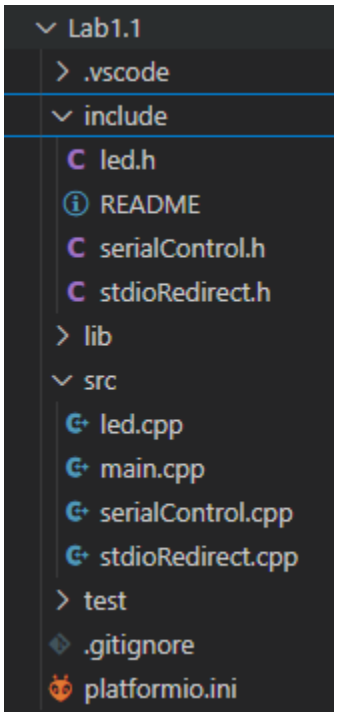

Figure 3.1 Project organization

This header file, named *led.h*, defines the interface for controlling an LED. It declares four functions without providing their implementations:

1. **initializeLed(int pin)** – This function is intended to initialize the LED by specifying the pin number on the microcontroller to which the LED is connected.
2. **turnOnLed()** – This function will be used to turn the LED on.
3. **turnOffLed()** – This function will be used to turn the LED off.
4. **getLedState()** – This function returns a boolean value indicating the current state of the LED (e.g., true if the LED is on, false if it is off).

This header file, named *serialControl.h*, defines the interface for handling serial communication. It declares two functions:

1. **initializeSerial(unsigned long baudRate)** – This function is designed to initialize the serial communication by setting the baud rate, which determines the speed of data transmission between the microcontroller and a connected device (e.g., a computer or terminal).
2. **processSerialCommands()** – This function is intended to process incoming serial commands. It will likely be used to read data from the serial buffer, interpret the commands, and trigger corresponding actions (e.g., turning the LED on or off).

This header file, named *stdio_redirect.h*, defines the interface for redirecting standard input and output. It declares one function:

1. **initializeStdio()** – This function is designed to initialize the redirection of standard input and output streams, allowing the use of standard C functions like printf() and scanf() with the serial interface on the microcontroller.

This implementation file, named *led.cpp*, provides the functionality for controlling an LED connected to an Arduino. It includes the following key elements:

1. **Static Variables**:

- ledPin: Stores the pin number to which the LED is connected. It is marked as static, making it local to this file.
- ledState: Tracks the current state of the LED (true for ON, false for OFF), also defined as static to maintain its state within this file.

2. **Function Implementations**:
   - **initializeLed(int pin)**:
     - Initializes the LED by setting the specified pin as an output.
     - Ensures the LED starts in the OFF state by writing LOW to the pin.
   - **turnOnLed()**:
     - Sets the pin to HIGH, turning the LED on.
     - Updates ledState to true.
   - **turnOffLed()**:
     - Sets the pin to LOW, turning the LED off.
     - Updates ledState to false.
   - **getLedState()**:
     - Returns the current state of the LED (true for ON, false for OFF).

The file includes "led.h" for function declarations and <Arduino.h> to access Arduino-specific functions such as pinMode() and digitalWrite(). The use of static variables ensures that the state and pin configuration are preserved across function calls but remain inaccessible from other translation units, maintaining encapsulation.

The file *stdio_redirect.cpp* handles redirecting standard input and output to the Arduino's serial interface, enabling the use of printf() and scanf() functions for serial communication.

1. **Purpose**:
   - Redirects stdout to Serial.write() so that printf() can be used for output.
   - Redirects stdin to Serial.read() so that scanf() can be used for input.
2. **Functionality**:
   - **uart_putchar()**: A static function that sends characters to the serial output.
   - **uart_getchar()**: A static function that reads characters from the serial input, blocking until data is available.

- **initializeStdio()**:
  - Sets up custom FILE streams for both stdout and stdin.
  - Uses fdev_setup_stream() to link the streams to the corresponding functions.
  - Redirects the global stdout and stdin pointers to the newly configured streams.

3. **Usage**:
   - This allows the rest of the code to use standard C functions like printf() and scanf() for serial communication, improving readability and portability.

The file *serialControl.cpp* manages the serial communication, processing user commands to control the LED.

1. **Purpose**:
   - Initializes the serial communication.
   - Processes incoming commands to turn the LED on or off.
2. **Functionality**:
   - **initializeSerial(unsigned long baudRate)**:
     - Sets up the serial communication with the specified baud rate.
     - Displays available commands to the user.
   - **processSerialCommands()**:
     - Reads user input using scanf() with a buffer size of 32 characters.
     - Parses the input to check for the command structure: "led on" or "led off".
     - Calls turnOnLed() or turnOffLed() from led.cpp accordingly.
     - Displays confirmation or error messages using printf().
3. **Command Processing**:
   - The system recognizes the following commands:
     - "led on": Turns the LED on and confirms with a message.
     - "led off": Turns the LED off and confirms with a message.
     - Any other input results in an error message.
4. **Buffer Management**:

- Uses a BUFFER_SIZE of 32 to ensure safe handling of input strings, preventing buffer overflows.

This, *main.cpp*, is the entry point of the application, coordinating the initialization and execution of the system.

1. **Purpose**:
   - Initializes the hardware and software modules.
   - Continuously processes user commands in the main loop.
2. **Functionality**:
   - **setup()**:
     - Initializes the LED by specifying the pin (LED_PIN 13).
     - Sets up serial communication at 9600 baud.
     - Initializes the standard input and output redirection.
   - **loop()**:
     - Continuously checks for and processes serial commands using processSerialCommands().
3. **System Flow**:
   - On startup, the system displays available commands.
   - It then waits for user input and processes the commands accordingly, controlling the LED's state.

Results

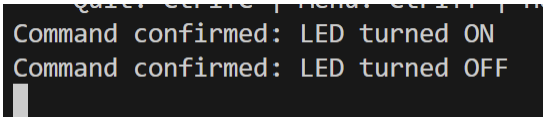

Figure 4.1 Console results

Figure 4.2 Physical result (led on)

Figure 4.3 Physical result (led of)

Conclusions

1. System Performance Analysis

The implemented system demonstrated efficient and stable operation in terms of serial communication and LED control. By using the STDIO library for data exchange via the serial interface, command processing is fast and accurate. Performance can be evaluated based on several factors:

- **Response Time:** Commands are processed almost instantly, with minimal latency determined only by the serial interface speed (9600 baud).
- **Reliability:** The system correctly handles the commands "led on" and "led off," providing appropriate feedback to the user.
- **Scalability:** The modular structure allows the system to be extended by adding other peripherals, such as buttons, LCD displays, or sensors.

Identified Limitations

Although the system operates as expected, there are some limitations that could be improved:

1. **Lack of Robust Command Validation** – Currently, the program only checks for exact commands ("led on" and "led off") without handling input errors, such as extra spaces or typos.
2. **Command Line Interface Limitation** – The user must manually enter commands through a serial terminal. A more user-friendly alternative would be a graphical user interface (GUI) for controlling the LED.
3. **Lack of Additional Visual Confirmation** – In real-world applications, it would be useful for the system to provide visual or audio confirmation for each action.

2. Conclusions on Obtained Results

After completing the laboratory work, the following results were achieved:

- A functional serial interface was successfully implemented, receiving and processing text commands from the user.
- The LED is controlled efficiently, and the user receives confirmation messages after each action.
- A modular architecture was used, ensuring code clarity and reusability.
- The implementation was tested on both the Arduino Mega 1280 board and in simulators to verify correct functionality.

This work demonstrated the importance of separating the interface from the implementation by using .h and .cpp files, allowing for good project organization.

3. Impact of the Technology Used in Real-World Applications

The technology used in this project has numerous practical applications, serving as an example of serial interface control. This approach is used in many embedded and industrial systems:

- **Industrial Automation:** Many factory devices are controlled via serial interfaces, allowing communication between machines and central systems.
- **IoT Devices:** The serial interface is used for configuring and controlling IoT devices, enabling remote monitoring.
- **Robotics and Embedded Systems:** Serial commands are essential for teleoperating robots, allowing control from a PC or main microcontroller.
- **Smart Lighting Control:** The same technique can be used for lighting control in smart homes, by turning lights on and off based on user commands.

In conclusion, this laboratory work represents a fundamental example of peripheral control through the serial interface, with applications in IoT, automation, and embedded electronics.

4. Improvement Suggestions

To enhance the project, the following optimizations could be implemented:

1. **Improved Command Validation** – Introducing more robust validation to allow alternative variations of commands (e.g., "LED ON," "Led On," etc.).

2. **Interface Expansion** – Developing a graphical user interface (GUI) for users using Python and Tkinter or a web application.
3. **Adding a Physical Button** – Allowing the LED to be turned on/off not only through the serial interface but also with a button connected to the microcontroller.
4. **Using an LCD Display** – Displaying the LED's status on an LCD screen for visual confirmation.
5. **Wi-Fi Network Control** – Using an ESP32 microcontroller to allow LED control via a web page.

By implementing these improvements, the project could become more flexible, scalable, and user-friendly in real-world applications.

Note on AI Tool Usage

During the drafting of this report, the author used ChatGPT for generating and structuring the content. The resulting information was reviewed, validated, and adjusted according to the requirements of the laboratory work, ensuring technical accuracy and clarity of explanations. The use of this AI tool was aimed at structuring and optimizing the presentation of information without replacing personal analysis and understanding of the subject.

Bibliography

1. Official Arduino Documentation
    - Arduino Reference – Serial Communication https://www.arduino.cc/reference/en/#communication
    - Arduino Mega 1280 Pinout & Datasheet https://docs.arduino.cc/hardware/mega-1280
2. PlatformIO Official Documentation
    - PlatformIO for Arduino Development https://docs.platformio.org/en/latest/platforms/atmelavr.html
3. TUM Courses

- Introducere în Sistemele Embedded și Programarea Microcontrolerelor
- Principiile comunicației seriale și utilizarea interfeței UART

Appendix

1. **GitHub**: https://github.com/KaBoomKaBoom/ES_Labs.git

2. **led.h**

```
// led.h
#ifndef LED_H
#define LED_H

void initializeLed(int pin);
void turnOnLed();
void turnOffLed();
bool getLedState();

#endif
```

3. **serialControl.h**

```
// serialControl.h
#ifndef SERIAL_CONTROL_H
#define SERIAL_CONTROL_H

void initializeSerial(unsigned long baudRate);
void processSerialCommands();

#endif
```

4. **stdioRedirect.h**

```
// stdio_redirect.h
#ifndef STDIO_REDIRECT_H
#define STDIO_REDIRECT_H
```

```
#include <stdio.h>

void initializeStdio();

#endif
```

5. led.cpp

```
// led.cpp (unchanged)
#include "led.h"
#include <Arduino.h>

static int ledPin;
static bool ledState = false;

void initializeLed(int pin) {
   ledPin = pin;
   pinMode(ledPin, OUTPUT);
   digitalWrite(ledPin, LOW);
}

void turnOnLed() {
   digitalWrite(ledPin, HIGH);
   ledState = true;
}

void turnOffLed() {
   digitalWrite(ledPin, LOW);
   ledState = false;
}

bool getLedState() {
```

```
    return ledState;
}
```

6. main.cpp

```
// main.cpp
#include <Arduino.h>
#include "led.h"
#include "serialControl.h"
#include "stdioRedirect.h"

#define LED_PIN 13
#define BAUD_RATE 9600

void setup() {
    initializeLed(LED_PIN);
    initializeSerial(BAUD_RATE);
    initializeStdio();  // Initialize stdio redirection
}

void loop() {
    processSerialCommands();
}
```

7. serialControl.cpp

```
// serialControl.cpp
#include "serialControl.h"
#include "led.h"
#include <Arduino.h>
#include <stdio.h>

#define BUFFER_SIZE 32
```

```
void initializeSerial(unsigned long baudRate) {
   Serial.begin(baudRate);
   while (!Serial) {
      ; // Wait for serial port to connect
   }

   printf("LED Control System Ready\n");
   printf("Available commands:\n");
   printf("- led on  : Turn on the LED\n");
   printf("- led off : Turn off the LED\n");
}

void processSerialCommands() {
   char command[BUFFER_SIZE];

   if (scanf("%31s", command) == 1) {  // Read up to 31 chars to leave room for null
terminator
      char nextWord[BUFFER_SIZE];
      if (scanf("%31s", nextWord) == 1) {
         if (strcmp(command, "led") == 0) {
            if (strcmp(nextWord, "on") == 0) {
               turnOnLed();
               printf("Command confirmed: LED turned ON\n");
            }
            else if (strcmp(nextWord, "off") == 0) {
               turnOffLed();
               printf("Command confirmed: LED turned OFF\n");
            }
            else {
               printf("Unknown LED command: %s\n", nextWord);
            }
```

```
            }
            else {
                printf("Unknown command: %s %s\n", command, nextWord);
            }
        }
    }
}
```

8. stdioRedirect.cpp

```
// stdio_redirect.cpp
#include "stdioRedirect.h"
#include <Arduino.h>

// Redirect stdout to Serial
static int uart_putchar(char c, FILE* stream) {
    Serial.write(c);
    return 0;
}

// Redirect stdin to Serial
static int uart_getchar(FILE* stream) {
    while (!Serial.available());
    return Serial.read();
}

void initializeStdio() {
    // Set up stdout and stdin
    static FILE uart_stdout;
    static FILE uart_stdin;

    fdev_setup_stream(&uart_stdout, uart_putchar, NULL, _FDEV_SETUP_WRITE);
```

```
    fdev_setup_stream(&uart_stdin, NULL, uart_getchar, _FDEV_SETUP_READ);

    stdout = &uart_stdout;
    stdin = &uart_stdin;
}
```