Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 4.1
Discipline: Embedded Systems
Topic: Actuators with Binary Interface - Relay Control (Serial/Keypad and LCD).

Done by: Musin Vladislava, st.gr. FAF-222

Checked by: asist.univ. Martîniuc Alexei

Chișinău 2025

1. Analysis of the situation in the field

Description of the technologies used and the context of the developed application

The developed application focuses on relay-based actuator control using microcontrollers, particularly within embedded systems. The primary technology utilized is microcontroller-based automation, which allows for the efficient control of electrical loads with minimal user intervention. The application is designed to manage high-power devices through low-power control circuits, ensuring safety and reliability in industrial, home automation, and IoT applications. The relay acts as an electrically operated switch, making it a crucial component in systems where isolation between control logic and the power circuit is required.

Another key technology in this application is serial communication, facilitated by the microcontroller's UART interface. This enables seamless user interaction through a terminal interface, allowing commands to be sent via a computer or external controller. The combination of embedded programming, hardware interfacing, and digital control techniques ensures that the system is both flexible and scalable, making it applicable in various real-world automation scenarios.

Presentation of the hardware and software components used

Microcontroller (Arduino Mega 2560)

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560 chip. It is chosen for this application due to its multiple I/O pins, UART communication capabilities, and robust processing power. The microcontroller serves as the core processing unit, reading input commands from the serial interface and actuating the relay accordingly.

Relay Module

A relay is an electromechanical switch used to control high-power loads using low-power signals. It is utilized in this application to isolate the control circuit from the load circuit, ensuring safety and efficient switching of external devices like motors or lamps.

Diode (1N4007)

A 1N4007 diode is connected across the relay coil to protect the circuit from voltage spikes caused by the inductive nature of the relay coil. This flyback diode prevents potential damage to the microcontroller and other sensitive components.

Explanations on the system architecture and justification of the choice of solution

The system architecture follows a modular design, comprising three main layers: the input interface (serial communication), the processing unit (microcontroller), and the output interface (relay-controlled load). The microcontroller acts as the central processing unit, receiving commands from a computer terminal via UART. These commands dictate the activation or deactivation of the relay, which in turn controls an external electrical device.

The choice of this architecture is justified by its flexibility, scalability, and safety. Using a microcontroller ensures programmable control, enabling dynamic modification of the system's behavior. The relay provides electrical isolation, reducing risks associated with high-power circuit switching. Additionally, serial communication offers a simple yet effective means of user interaction without requiring additional hardware such as an LCD or keypad.

Use case

The primary use case for this system is automated actuator control in smart home applications. For example, the relay can be used to control household appliances such as lights, fans, or heating systems based on remote commands sent via a serial interface. The user can turn devices on or off without physical interaction, improving convenience and energy efficiency.

Another practical application is industrial automation, where the relay system can be employed to control machinery or safety mechanisms based on sensor inputs. By integrating the relay module with additional components such as temperature sensors, the system can automate cooling systems in industrial environments, triggering fans or alarms when predefined thresholds are exceeded. This enhances system responsiveness and reduces the need for manual intervention.

2. Design

The purpose of the work:

Creating a modular application for a microcontroller (MCU) to control binary actuators (light bulb or fan) using commands received through the STDIO interface, either through a serial terminal or through a keyboard (Keypad), and to report the actuator status to an LCD display or through the serial interface.

Objectives:

- Familiarization with the principles of operation and control of relays in embedded applications.
- Using the STDIO library for user interaction via the serial terminal and/or matrix keyboard (Keypad).
- Implementing binary control of actuators (light bulb or fan) via a relay.

- Developing a modular solution, structured on separate files for each peripheral, thus ensuring reuse in future projects.
- Documenting the software architecture and presenting block diagrams and electrical diagrams as an integral part of the design methodology.

Architectural Diagram

This architectural diagram outlines the interaction between software and hardware components in a system built around an Arduino Uno, designed to control a relay and an LED via serial commands. The diagram is divided into two main sections: "Computer Software Components" at the top and "Hardware Arduino Uno" at the bottom, connected by a "USB Cable," indicating the communication link between them.

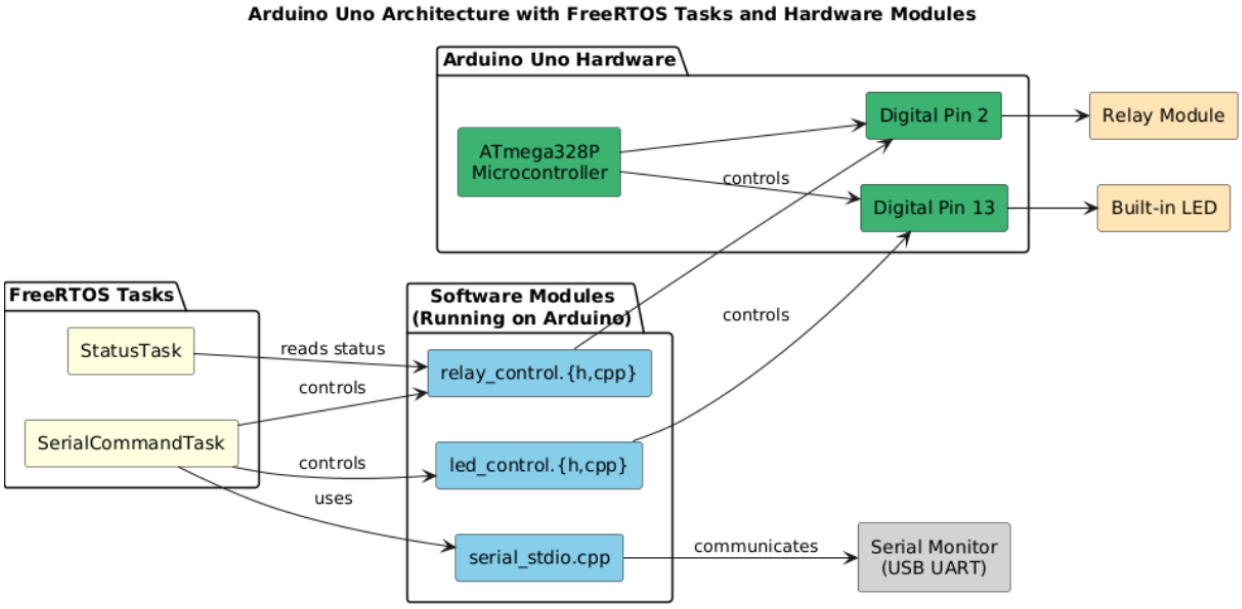


Figure 2.1. *General Architectural Diagram*

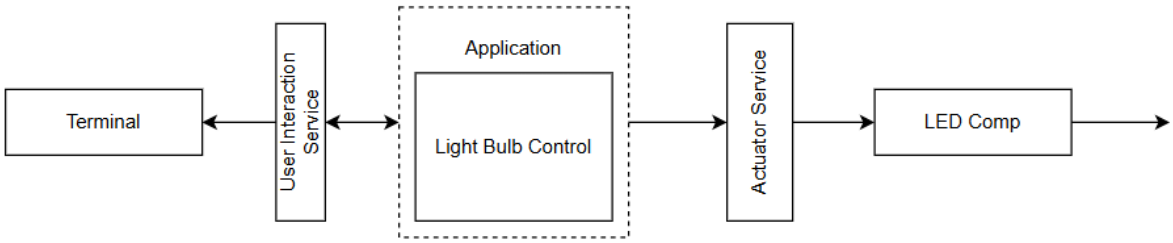
The "Computer Software Components" section depicts the software architecture, starting with "Main.cpp," which likely represents the main program file. It branches into two primary tasks: "Status Task" and "Serial Command Task," suggesting a multi-threaded or event-driven system. These tasks interact with several modules: "Serial STDIO" for serial communication, "Relay Control Module" for managing the relay, and "LED Control Module" for managing the LED. The "Serial STDIO" module communicates with the Arduino Uno via the USB cable, enabling data exchange between the computer and the microcontroller.

The "Hardware Arduino Uno" section details the hardware components. It starts with the "USB/Serial Port," which receives commands from the computer. The "ATmega328P Microcontroller" is the core of the Arduino, processing these commands. It interacts with "Digital Pin 2" and "Digital Pin 13," which are connected to a "Physical Relay Module" and a "Built-in LED," respectively. The relay and LED are the physical outputs that are controlled by the Arduino

based on commands received from the computer. This diagram illustrates a typical embedded system setup where a computer sends commands to a microcontroller via serial communication to control external hardware components.

The next diagram illustrates a system architecture focused on controlling an LED component, likely representing a light bulb, through user interaction. The system comprises several distinct components that interact to facilitate this control. The "Terminal" represents the user interface, where commands or instructions are input. This input is then processed by the "User Interaction Service," which acts as an intermediary, translating user commands into a format understandable by the application. The core of the system is the "Application," specifically the "Light Bulb Control" module, which contains the logic for managing the LED's state.

The "Light Bulb Control" module communicates with the "Actuator Service," which is responsible for translating the application's commands into signals that directly control the physical LED component ("LED Comp"). This service acts as an abstraction layer, hiding the specifics of the hardware interface from the application. The LED component itself is the final stage, receiving signals from the "Actuator Service" to change its state (e.g., on/off, brightness). The unidirectional arrows indicate the flow of information and control from the user input at the "Terminal" to the final output at the "LED Comp," highlighting the system's linear, sequential processing of control commands.



. ***Figure 2.3.*** *System Diagram*

Block diagrams

The following flowchart illustrates the operational flow of a system, likely an embedded system utilizing FreeRTOS, designed to control a relay and an LED via serial commands. The process begins with initialization, encompassing both serial communication and standard input/output (STDIO). Subsequently, two tasks are created: one for handling serial commands and another for managing system status. The FreeRTOS scheduler is then started, allowing these tasks to run concurrently.

The "Serial Command Task" specifically handles user input through serial communication. It starts by setting up the relay and LED, then prompts the user for a command and reads it. The

command is then parsed, branching into different actions based on the input. If the command is "relay on", the relay is turned on, and the LED is updated accordingly. Conversely, if the command is "relay off", the relay is turned off, and the LED is updated. Other commands include "status" to display the current system status, "help" to display available commands, and "other" to handle invalid commands, resulting in an error message. After processing each command, a 10ms delay is introduced before looping back to read the next command.

The "Status Task" operates in parallel, focusing on monitoring and reporting the relay status. It begins by retrieving the current relay status, which is then printed to the serial output. Following this, a 5000ms delay is implemented before the task loops back to retrieve and print the status again. This task essentially provides periodic updates on the relay's state, independent of the user commands processed by the "Serial Command Task". The concurrent execution of these tasks, managed by FreeRTOS, allows for responsive command handling and continuous status monitoring.

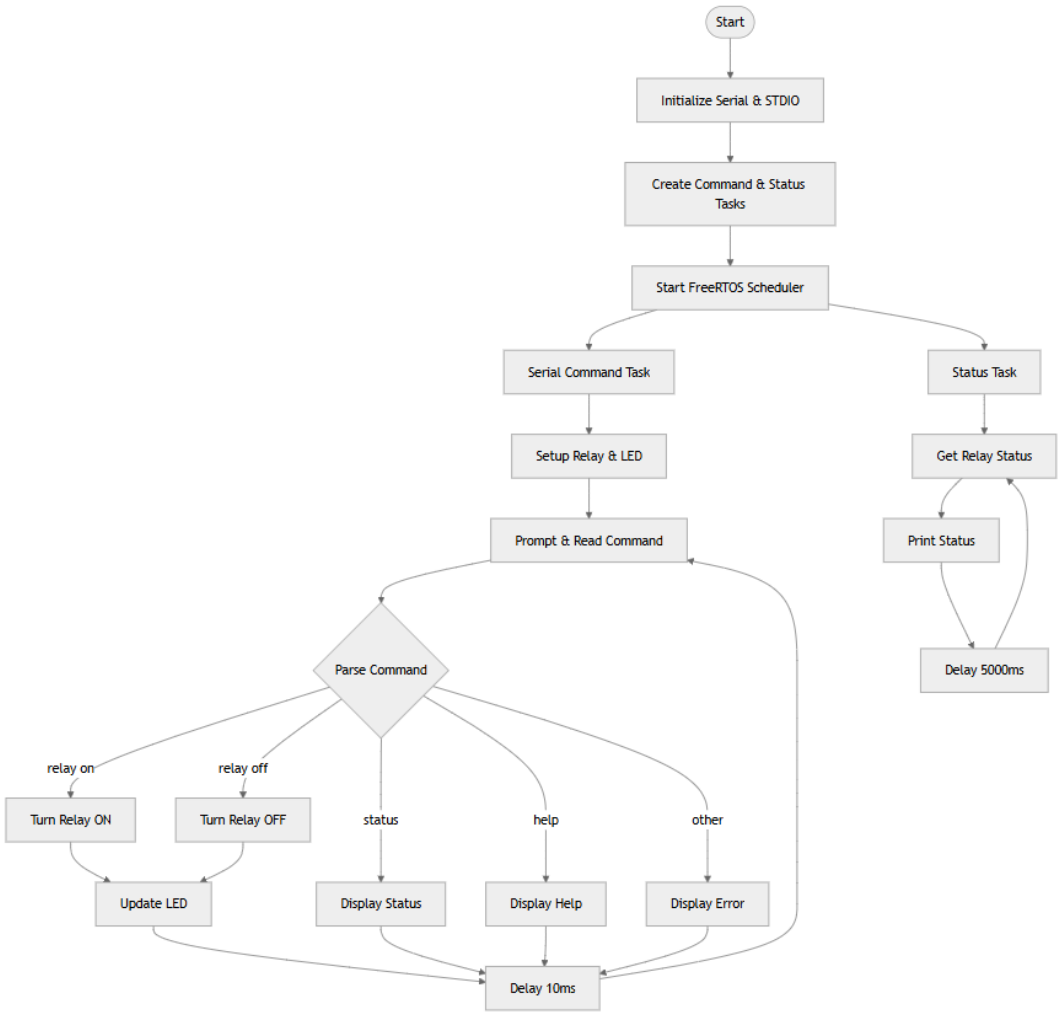


Figure 2.4. *Flowchart Diagram*

Electrical Schematic

In this electrical schematic, an Arduino Uno microcontroller is the central component, controlling both an LED and a relay. The Arduino's digital pin D13 is connected to an LED labeled "D1 RED", suggesting that the LED will light up when D13 is set to HIGH. Additionally, digital pin D8 is connected to the "COM1" terminal of a relay labeled "K1". The relay's "COIL1" and "COIL2" terminals are connected to the power supply, implying that when D8 is activated, the relay coil will be energized, switching the relay's contacts.

The schematic also shows the power connections for the Arduino and the relay. Both the Arduino and the relay are supplied with 5V from "U1_5V". The ground connections for both components are tied to "U1_GND", ensuring a common ground reference. This setup suggests that the Arduino is programmed to control the LED and the relay based on its input or programmed logic. The relay's "COM2", "NC", and "NO" terminals are available for connecting external devices, allowing the Arduino to switch higher-power circuits or devices through the relay's contacts. The absence of other components suggests this is a basic setup for demonstrating or testing the Arduino's digital output capabilities and relay control.

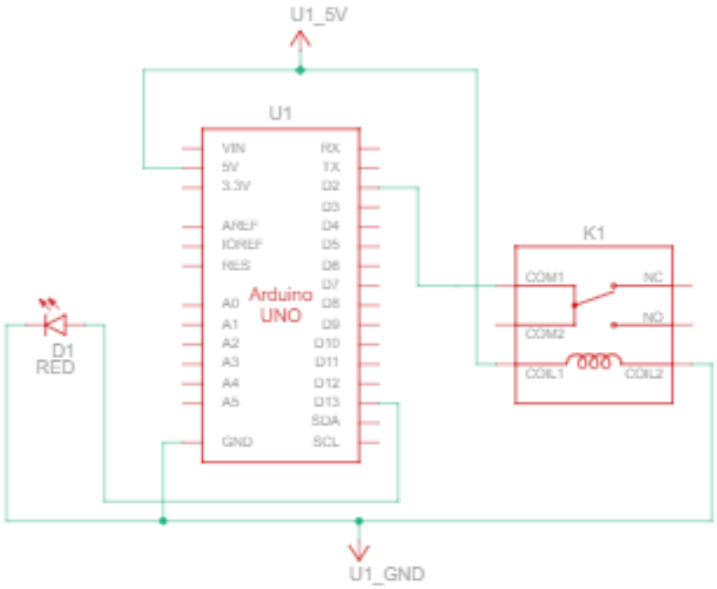


Figure 2.5. *Electrical Schematic*

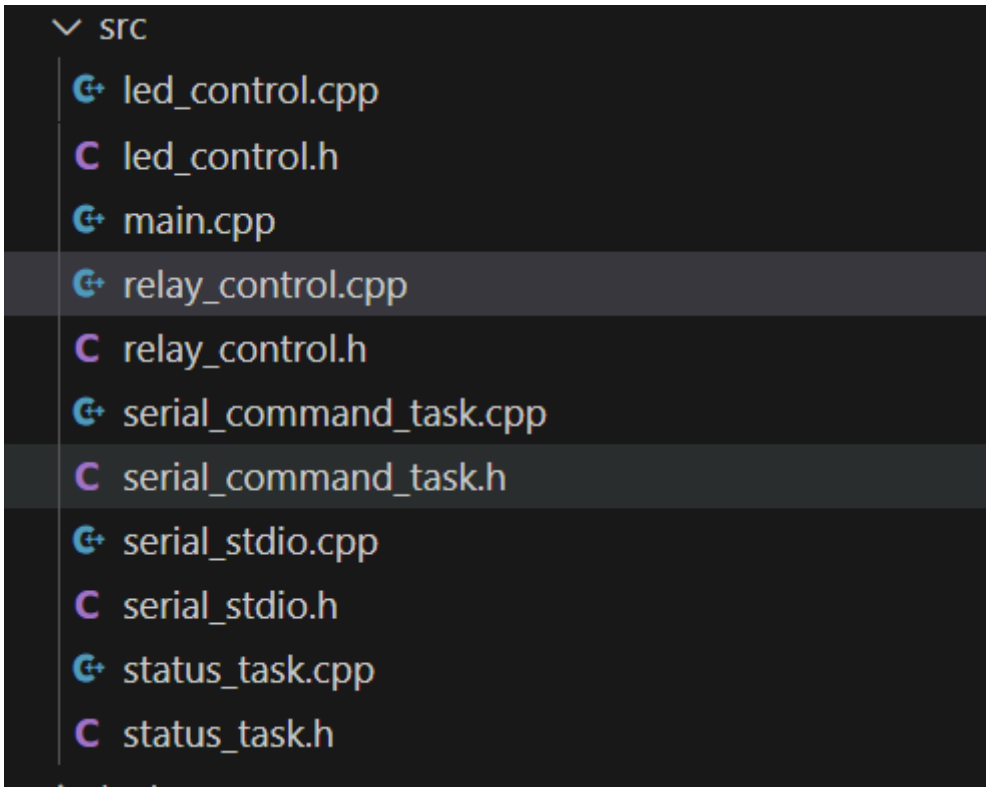
Additional Functionality

To improve user guidance and interaction with the relay control system, we introduced a help_detail function. Unlike the standard help command, which provides a brief list of available commands, help_detail allows users to request specific information about a command's functionality and usage. For example, by entering help relay on, users receive a detailed explanation of how the command activates the relay, its expected response, and any safety

considerations. This function enhances the serial interface by providing a more intuitive and informative user experience, ensuring clarity for users unfamiliar with the system's capabilities.

The help_detail function works by parsing the user's input and checking if they have requested additional details on a known command. If a specific command is provided (e.g., help status), the system prints a more in-depth description of what the command does and how it behaves in different scenarios. If no valid command is specified, the function defaults to displaying the general help menu. This modular approach improves usability while keeping the core command structure simple and efficient within the FreeRTOS-based environment.

Project Structure



Modular Implementation Details

- *Main Application*

This module serves as the entry point for the Arduino FreeRTOS application, responsible for initializing the system and launching the task scheduler. It sets up serial communication, initializes the serial stdio interface for printf/scanf functionality, and creates two tasks: SerialCommandTask for handling user input and StatusTask for periodically reporting system status.

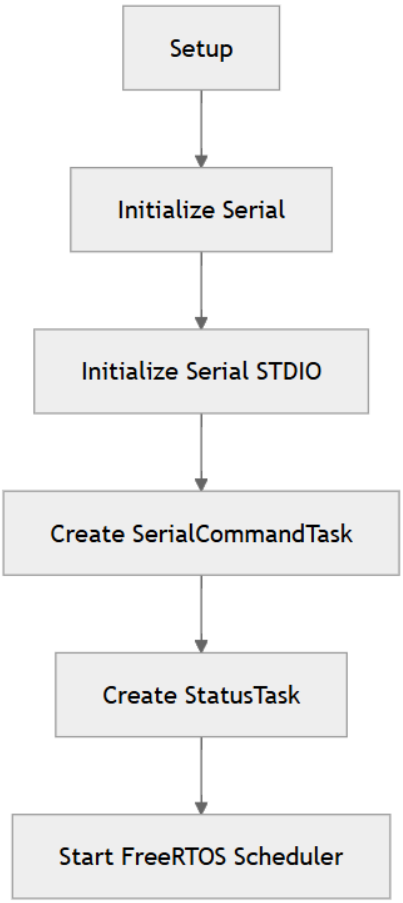


Figure 2.6. *Main Application Block Diagram*

The main module demonstrates proper FreeRTOS initialization by creating tasks with appropriate stack sizes and priorities before starting the scheduler. It provides a simple user interface by prompting for commands and doesn't use the traditional Arduino loop() function since FreeRTOS takes control of task scheduling after initialization.

- *serial_command_task Module*

This module implements a command-line interface that processes user input to control the relay and LED components. It runs as a high-priority task that continuously prompts for and processes commands like "relay on", "relay off", "status", and "help", translating these text commands into actions on the physical hardware.

The task demonstrates command parsing techniques using string comparison and implements a simple state machine for controlling hardware. It properly initializes the relay and LED hardware on startup and includes a 10ms delay at the end of each loop iteration to prevent task starvation, allowing other tasks to execute.

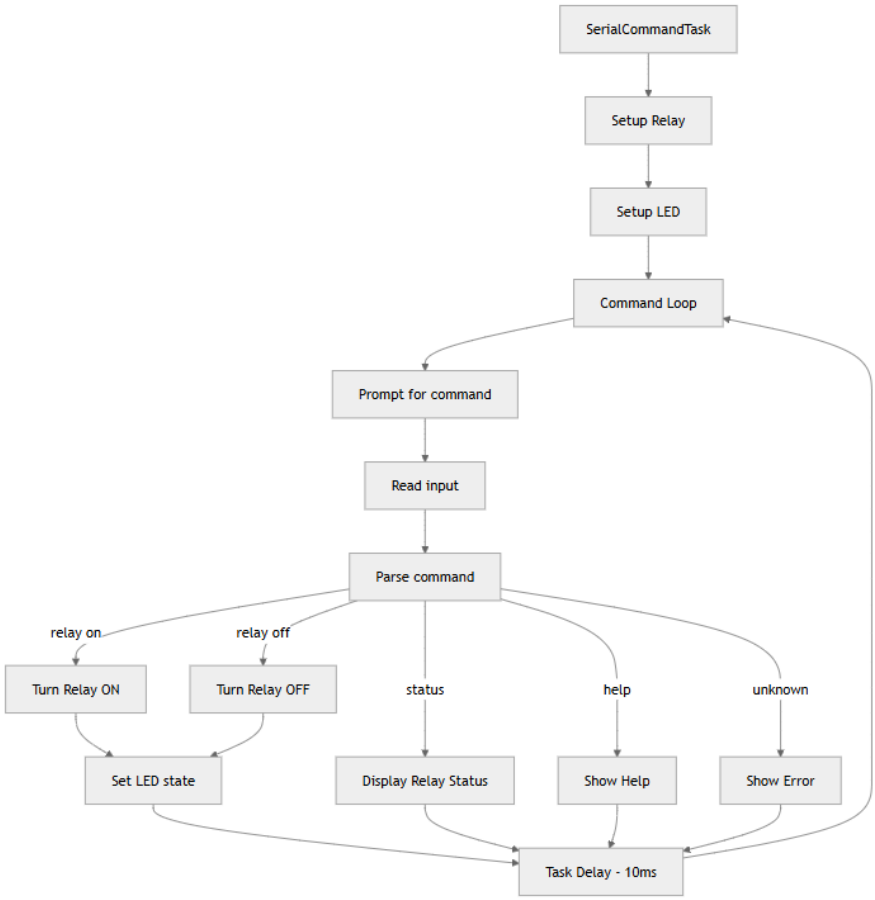


Figure 2.7. *serial_command_task Module Block Diagram*

- *status_task Module*

This module implements a low-priority background task that periodically reports the system status, including the current state of the relay. It runs independently of user interaction, providing automatic status updates every 5 seconds to help monitor the system's operation without requiring manual status checks.

The status task demonstrates the use of vTaskDelay for non-blocking delays in a FreeRTOS environment and showcases how independent tasks can share access to hardware state information through the relay control module. Its simple implementation highlights FreeRTOS's ability to handle periodic background processes while other user-facing tasks continue to run.

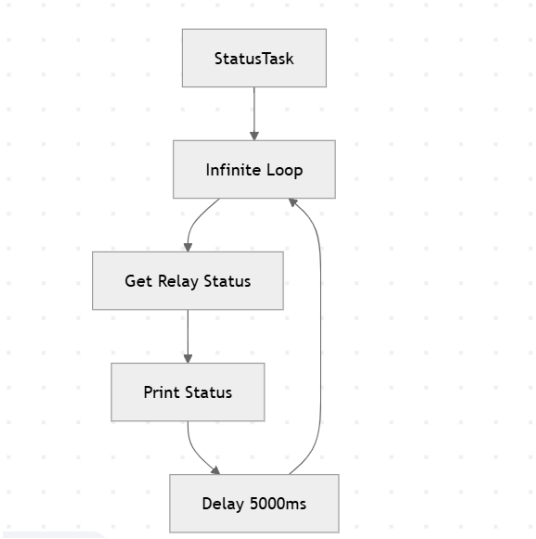


Figure 2.8. *status_task Module Block Diagram*

- *serial_stdio Module*

This module provides a bridge between Arduino's Serial interface and standard C input/output functions, allowing the application to use familiar functions like printf(), fgets(), and scanf(). It implements custom stream handlers that map stdin and stdout to the Arduino Serial port, enabling more natural text-based interactions with users.

Figure 2.9. *serial_stdio Module Block Diagram*

The implementation includes callback functions for handling character input and output through the serial port, demonstrating how to integrate standard C I/O functions with microcontroller platforms. This abstraction simplifies text processing throughout the application and makes the codebase more portable and readable.

- *led_control Module*

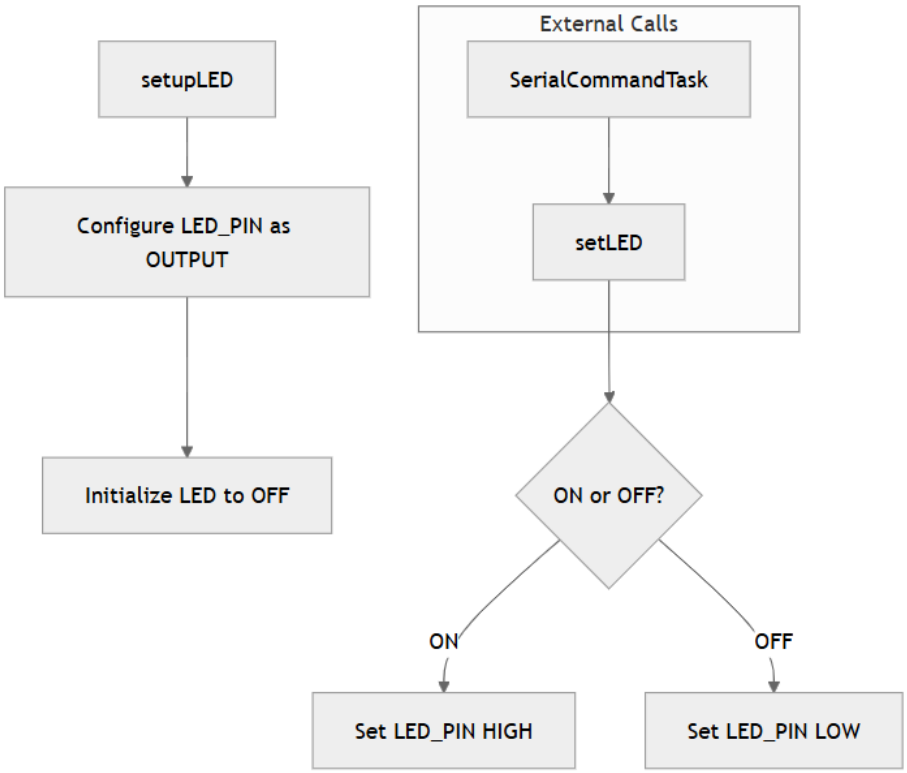


Figure 2.10. *led_control Module Block Diagram*

This module provides a simple abstraction for controlling the onboard LED, typically connected to pin 13 on many Arduino boards. It encapsulates the initialization and state management of the LED, providing a clean interface that hides the low-level pin manipulation details from other parts of the application.

The implementation demonstrates proper hardware abstraction by isolating pin definitions and offering a boolean interface for controlling the LED state. This module works in tandem with the relay control, as the LED serves as a visual indicator of the relay's current state, helping users visually confirm system operation without checking the serial output.

- *relay_control Module*

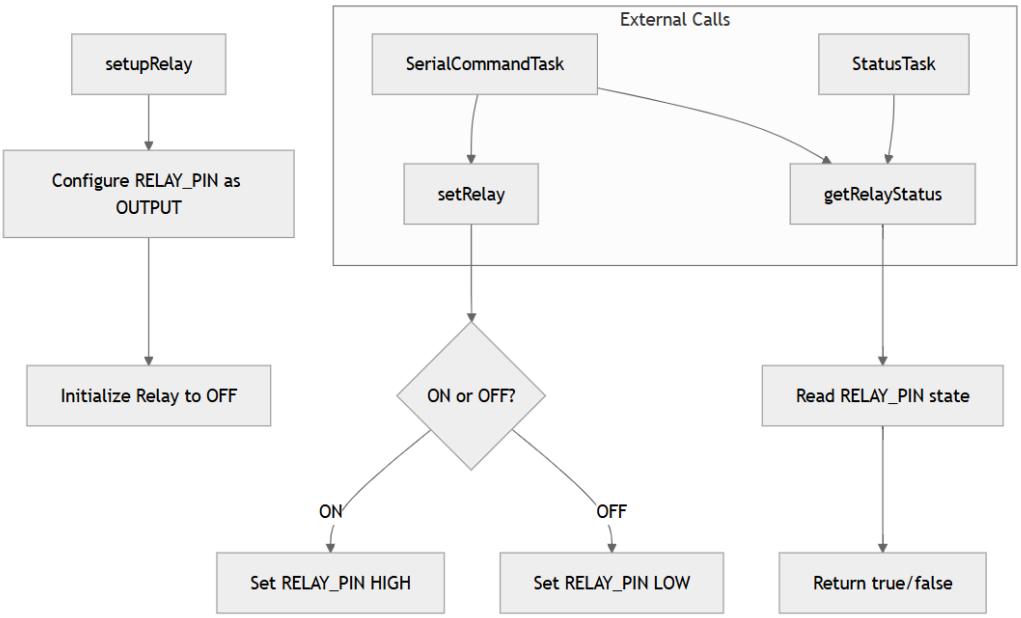
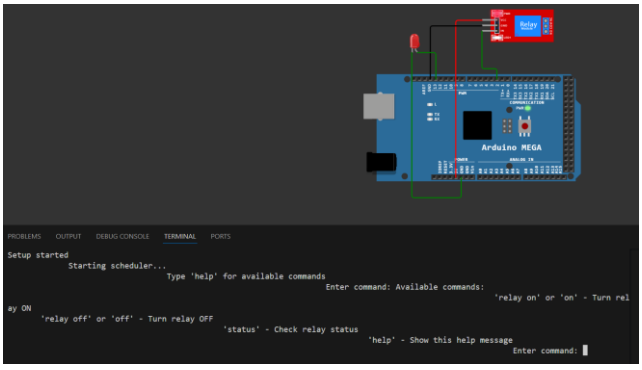


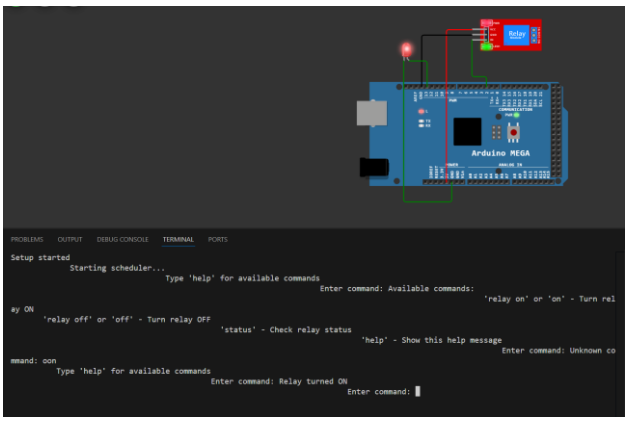
Figure 2.11. *relay_control Module Block Diagram*

This module manages the physical relay hardware connected to the Arduino, providing functions to initialize, control, and query the relay's state. It abstracts the digital I/O operations required to control the relay, making the relay appear as a simple on/off device to the rest of the application.

The implementation follows a clean hardware abstraction pattern with dedicated setup, control, and status query functions. It maintains the relay's state internally while providing a getRelayStatus function that allows other modules to check the relay condition without directly accessing hardware pins, demonstrating good separation of concerns and module encapsulation.

Results Presentation





Conclusions

In this laboratory exercise, we successfully implemented a FreeRTOS-based control system for a relay actuator using the Arduino Uno. Through the serial interface, we issued commands such as "on", "off", and "status" to control the relay and observe its behavior in real time. The integration of printf() and scanf() for standard input/output allowed us to interact with the system efficiently, simulating a command-line interface. Additionally, an LED indicator was used to visually represent the relay's state, improving system feedback and usability.

This lab reinforced key embedded systems concepts, such as task scheduling with FreeRTOS, GPIO-based actuator control, and structured software layering. By separating hardware abstraction (MCAL) from application logic, we created a modular and maintainable design. The experiment highlighted the importance of real-time responsiveness and modular code organization when working with hardware interfaces. Overall, this lab provided valuable hands-on experience with embedded multitasking and real-time control of digital actuators.

AI Tool Usage Note

During the writing of this report, the author used ChatGPT to consolidate the content. The resulting information was reviewed, validated and adjusted according to the requirements of the laboratory work.

References

1. Arduino Tutorial. Arduino grows up and learns to talk! https://www.ladyada.net/learn/arduino/lesson4.html
2. SRI-25-Lab 4.1: Actuatori - Actuator binar (Releu - FAN) https://www.youtube.com/watch?v=SvYX39XgKaQ

3. Arduino Serial Communication https://www.instructables.com/Arduino-Serial-Communication/
4. ArduinoDocs. https://docs.arduino.cc/tutorials/4-relays-shield/4-relay-shield-basics/
5. LastMinuteEngineers. https://lastminuteengineers.com/one-channel-relay-module-arduino-tutorial/

Appendix - Source Code

Main.cpp

```
#include <Arduino_FreeRTOS.h>
#include <Arduino.h>
#include "serial_stdio.h"
#include "serial_command_task.h"
#include "status_task.h"

TaskHandle_t serialTaskHandle = NULL;
TaskHandle_t statusTaskHandle = NULL;

void setup() {
  Serial.begin(9600);
  while (!Serial);

  initSerialSTDIO();
  printf("Setup started\n");

  xTaskCreate(SerialCommandTask, "SerialCmd", 192, NULL, 2, &serialTaskHandle);
  xTaskCreate(StatusTask, "Status", 128, NULL, 1, &statusTaskHandle);

  printf("Starting scheduler...\n");
  printf("Type 'help' for available commands\n");

  vTaskStartScheduler();
  printf("Scheduler failed to start!\n");
}

void loop() {
  // Not used with FreeRTOS
}
```

serial_command_task.h

```
#ifndef SERIAL_COMMAND_TASK_H
#define SERIAL_COMMAND_TASK_H

void SerialCommandTask(void *pvParameters);

#endif
```

serial_command_task.cpp

```
#include <Arduino_FreeRTOS.h>
#include "task.h"
#include <Arduino.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "relay_control.h"
#include "led_control.h"

void SerialCommandTask(void *pvParameters) {
  (void) pvParameters;

  setupRelay();
  setupLED();

  char command[50];

  for (;;) {
    printf("Enter command: ");
```

```
    if (fgets(command, sizeof(command), stdin) != NULL) {
      size_t len = strlen(command);
      if (len > 0 && command[len - 1] == '\n') {
        command[len - 1] = '\0';
      }

      for (size_t i = 0; i < len; i++) {
        command[i] = tolower(command[i]);
      }

      if (strcmp(command, "relay on") == 0 || strcmp(command, "on") ==
0) {
        setRelay(true);
        setLED(true);
        printf("Relay turned ON\n");
      }
       else if (strcmp(command, "relay off") == 0 || strcmp(command,
"off") == 0) {
        setRelay(false);
        setLED(false);
        printf("Relay turned OFF\n");
      }
      else if (strcmp(command, "status") == 0) {
         printf("Relay is currently %s\n", getRelayStatus() ? "ON" :
"OFF");
      }
      else if (strcmp(command, "help") == 0) {
        printf("Available commands:\n");
        printf("  'relay on' or 'on' - Turn relay ON\n");
        printf("  'relay off' or 'off' - Turn relay OFF\n");
        printf("  'status' - Check relay status\n");
        printf("  'help' - Show this help message\n");
      }
      else if (strcmp(command, "panic") == 0) {
        printf("⚠  System warning: Relay overload detected!\n");
      }
      else if (strlen(command) > 0) {
        printf("Unknown command: %s\n", command);
        printf("Type 'help' for available commands\n");
      }
    }

    vTaskDelay(pdMS_TO_TICKS(10)); // <--- This now works
  }
}
```

status_task.h

```
#ifndef STATUS_TASK_H
#define STATUS_TASK_H

void StatusTask(void *pvParameters);

#endif
```

status_task.cpp

```
#include <Arduino_FreeRTOS.h>
#include "task.h"
```

```
#include <stdio.h>
#include "relay_control.h"

void StatusTask(void *pvParameters) {
  (void) pvParameters;

  for (;;) {
    printf("Status: Scheduler is running! Relay is %s\n",
           getRelayStatus() ? "ON" : "OFF");
    vTaskDelay(pdMS_TO_TICKS(5000));
  }
}
```

serial_stdio.h

```
#ifndef SERIAL_STDIO_H
#define SERIAL_STDIO_H

void initSerialSTDIO();

#endif
```

serial_stdio.cpp

```
#include <Arduino.h>
#include <stdio.h>

int serial_putc(char c, FILE* stream) {
  Serial.write(c);
  return c;
}

int serial_getc(FILE* stream) {
  while (!Serial.available());
  return Serial.read();
}

FILE serial_stdout;
FILE serial_stdin;

void initSerialSTDIO() {
  fdev_setup_stream(&serial_stdout, serial_putc, NULL,
_FDEV_SETUP_WRITE);
  fdev_setup_stream(&serial_stdin, NULL, serial_getc,
_FDEV_SETUP_READ);
  stdout = &serial_stdout;
  stdin = &serial_stdin;
}
```

led_control.h

```
#ifndef LED_CONTROL_H
#define LED_CONTROL_H

void setupLED();
void setLED(bool on);

#endif
```

led_control.cpp

```
#include <Arduino.h>
#include "led_control.h"

#define LED_PIN 13

void setupLED() {
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW);
}

void setLED(bool on) {
  digitalWrite(LED_PIN, on ? HIGH : LOW);
}
```

relay_control.h

```
#ifndef RELAY_CONTROL_H
#define RELAY_CONTROL_H

void setupRelay();
void setRelay(bool on);
bool getRelayStatus();

#endif
```

relay_control.cpp

```
#include <Arduino.h>
#include "relay_control.h"

#define RELAY_PIN 2

void setupRelay() {
  pinMode(RELAY_PIN, OUTPUT);
  digitalWrite(RELAY_PIN, LOW);
}

void setRelay(bool on) {
  digitalWrite(RELAY_PIN, on ? HIGH : LOW);
}

bool getRelayStatus() {
  return digitalRead(RELAY_PIN) == HIGH;
}
```

platformio.ini

```
[env:uno]
platform = atmelavr
board = uno
framework = arduino
monitor_speed = 9600
build_unflags = -std=gnu++11
build_flags = -std=c++17
lib_deps =
    marcoschwartz/LiquidCrystal_I2C@^1.1.4
    chris--a/Keypad@^3.1.1
    sadr0b0t/arduino-timer-api@^0.1.0
    feilipu/FreeRTOS@^11.1.0-3
    paulstoffregen/OneWire@^2.3.8
    milesburton/DallasTemperature@^4.0.4
```

```
lib_ignore =
    ArduinoOTA
```