Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Embedded Systems

Laboratory Work #3

Button Duration Monitor with Multitasking

Bare-Metal Non-Preemptive Scheduling & FreeRTOS

Author:
Sava Luchian
std. gr. FAF-233

Verified:
Alexei Martiniuc

Chisinau 2026

Technical Task

Purpose of the Laboratory

Familiarise students with the fundamental concepts of embedded operating systems – both bare-metal non-preemptive scheduling and FreeRTOS preemptive multitasking – by implementing a button duration monitoring application that executes three concurrent tasks, demonstrates inter-task synchronisation, and reports periodic statistics via STDIO.

Objectives

1. Understand bare-metal non-preemptive task scheduling using timer interrupts, context structs, and recurrence/offset tables.
2. Implement a FreeRTOS preemptive application with binary semaphores and mutexes.
3. Port the same application logic between bare-metal and RTOS with minimal code changes.
4. Demonstrate non-blocking task design (no busy-waiting) and safe shared-resource access.
5. Validate system behaviour through Wokwi simulation and serial output analysis.

Individual Task

Design and implement on an **Arduino Uno (ATmega328P)** a multitasking button monitoring application structured into at least three tasks:

- **Task 1** – Detect button press/release and measure press duration; signal **Green LED** for short press ($< 500\,\text{ms}$) and **Red LED** for long press ($\geq 500\,\text{ms}$).
- **Task 2** – On each press: increment counters, accumulate duration statistics, and blink the **Yellow LED** ($5\times$ for short, $10\times$ for long, 100 ms period).
- **Task 3** – Every 10 s: print total presses, short/long counts, average durations via Serial, then reset all counters.

Implement the application twice: as a **Part 1** bare-metal sequential scheduler and as a **Part 2** FreeRTOS preemptive system.

1 Domain Analysis

1.1 Technological Context and Application Domain

Multitasking is a foundational concept in embedded systems, enabling a single processor to interleave multiple logical activities efficiently. Embedded applications range from simple sensor-polling loops to safety-critical real-time systems that must respond to events within hard deadlines.

Two principal scheduling models exist in the field:

- **Bare-metal non-preemptive (cooperative) scheduling**: The processor runs a single loop; tasks are short C functions invoked at regular intervals by a timer-driven dispatcher. No task can interrupt another. This model, popularised by automotive standards such as OSEK-VDX, is predictable, has zero context-switch overhead, and avoids concurrency hazards entirely.
- **Real-Time Operating System (RTOS) preemptive scheduling**: The kernel manages a set of independent tasks with their own stacks and can suspend the running task at any time to let a higher-priority task run. FreeRTOS is the leading open-source RTOS for microcontrollers, providing task creation, semaphores, mutexes, and queues.

This laboratory explores both models side-by-side on the same application to illustrate the trade-offs in complexity, resource usage, and correctness guarantees.

1.2 Hardware Components and Justification

The Arduino Uno was selected for its broad compatibility with PlatformIO, the feilipu/FreeRTOS AVR port, and the Wokwi hardware simulator, allowing rapid iteration without physical hardware.

1.3 Software Components and Technologies

- **PlatformIO + VS Code**: Multi-environment project management. Two environments (`baremetal`, `freertos`) share the same source tree and use `build_src_filter` and `build_flags` to select the active variant.
- **Arduino Framework (avr-arduino 5.2.0)**: GPIO abstractions (`pinMode`, `digitalWrite`), `Serial`, and the `millis()` timer.
- **AVR-libc**: Timer1 register access (`TCCR1A/B`, `OCR1A`, `TIMSK1`), ISR macro for bare-metal scheduling.

Component	Qty	Role in the application
Arduino Uno (ATmega328P)	1	Main MCU; 16 MHz, 2 KB SRAM, 32 KB Flash. Runs both bare-metal and FreeRTOS firmware.
Push-button	1	User input; one leg to D2, other to GND (INPUT_PULLUP activated).
Green LED + 220 Ω	1	Visual signal for short press (<500 ms).
Red LED + 220 Ω	1	Visual signal for long press (≥ 500 ms).
Yellow LED + 220 Ω	1	Statistics blink feedback after each press.
Breadboard + jumper wires	–	Prototyping interconnection.
USB cable	1	5 V power supply and serial communication.

Table 1: Hardware components used in Lab 3

- **feilipu/FreeRTOS 11.1.0-3**: FreeRTOS kernel port for ATmega devices; provides task creation (`xTaskCreate`), binary semaphores (`xSemaphoreCreateBinary`), mutexes (`xSemaphoreCreateMutex`), and `vTaskDelayUntil`.
- **Wokwi Simulator**: Circuit + firmware simulation directly in VS Code; used for all functional testing.

1.4 Analysis of Existing Solutions

Button debouncing and duration measurement are well-understood problems with established software patterns. The key design decision in this laboratory is the *scheduling architecture*:

- **Arduino `millis()`-based polling** (no scheduler): Simple but merges all logic into one `loop()`, becoming unmanageable above a few tasks.
- **TaskScheduler library**: Open-source cooperative scheduler for Arduino; comparable in concept to Part 1 but library-dependent.
- **FreeRTOS**: Industry-standard RTOS used in production embedded products (ESP-IDF, AWS IoT, motor controllers). Provides all required synchronisation primitives and deterministic task wakeup.

The chosen implementation follows the course theory: a hand-crafted bare-metal scheduler (Part 1) demonstrates the principles, then FreeRTOS (Part 2) replaces the

manual machinery with proven OS infrastructure while keeping the application logic unchanged.

1.5 Real-World Applications

Non-preemptive task scheduling and RTOS-based embedded multitasking appear across a wide range of commercial and industrial products:

- **Industrial process controllers**: Use cooperative scheduling to sequence PLC-style operations with deterministic intervals and no complex synchronisation overhead. The producer-consumer task model used here mirrors standard IEC 61131 cyclic task concepts.
- **Automotive body control modules (BCM)**: OSEK/VDX-based non-preemptive kernels manage window, lock, and lighting tasks at fixed periods – the same recurrence/offset dispatch pattern implemented in Part 1.
- **Battery-powered sensor nodes**: Cooperative schedulers minimise background CPU activity and allow deep-sleep between task periods, extending battery life. The 20 ms polling cycle used for the button driver is a common IoT sensor pattern.
- **Medical device firmware**: Non-preemptive tasks with interrupt-only concurrency are favoured in safety-critical devices (IEC 62304) because their execution model is entirely auditable and testable without race-condition scenarios.
- **RTOS consumer electronics**: FreeRTOS runs in millions of IoT devices (ESP32, STM32-based products). The semaphore-driven event propagation from Task 1 to Task 2 mirrors the event-driven architecture of AWS IoT device SDKs built on FreeRTOS.
- **Home automation controllers**: Coordinate multiple peripherals (buttons, relays, sensors) using round-robin or priority task execution – exactly the three-task pattern demonstrated here.
- **Motor controller firmware**: Use `vTaskDelayUntil`-style periodic tasks for current-control loops at fixed frequencies, and event semaphores for fault handling – both patterns present in Part 2.

2 Conceptualisation and Design

2.1 Technical Requirements

ID	Category	Requirement	Type
R-F-01	Button	System shall detect button press and release with software debounce (stable for ≥ 20 ms).	Func.
R-F-02	Button	System shall measure press duration in milliseconds from first stable press to first stable release.	Func.
R-F-03	LED	Green LED shall illuminate for 800 ms after a short press (< 500 ms) is detected.	Func.
R-F-04	LED	Red LED shall illuminate for 800 ms after a long press (≥ 500 ms) is detected.	Func.
R-F-05	Statistics	System shall count total, short, and long presses independently and accumulate duration sums.	Func.
R-F-06	LED	Yellow LED shall blink 5 times (100 ms ON/OFF) after each short press.	Func.
R-F-07	LED	Yellow LED shall blink 10 times (100 ms ON/OFF) after each long press.	Func.
R-F-08	Report	System shall print a statistics report to Serial every 10 s: total, short, long counts and average durations.	Func.
R-F-09	Report	System shall reset all counters to zero immediately after printing each report.	Func.
R-F-10	Scheduling	Part 1 shall use a non-preemptive sequential scheduler driven by a 1 ms Timer1 ISR.	Func.
R-F-11	Scheduling	Part 2 shall use FreeRTOS preemptive tasks with binary semaphore and mutex synchronisation.	Func.
R-NF-01	Latency	Maximum latency from button edge to LED response shall be < 100 ms.	Non-func.
R-NF-02	CPU Load	CPU utilisation per system tick shall remain below 70%.	Non-func.
R-NF-03	Memory	Firmware shall fit within 32 KB Flash and 2 KB SRAM on the Arduino Uno.	Non-func.
R-NF-04	Modularity	Application shall be structured in dedicated modules (config, peripherals, shared, os, tasks).	Non-func.

R-NF-05	Portability	Core task logic shall be shared between bare-metal and FreeRTOS using conditional compilation only.	Non-func.

Table 2: Technical requirements for Lab 3

2.2 Architectural Design

2.2.1 System Structural Architecture

The application is divided into five software layers following the hardware–software boundary principle:

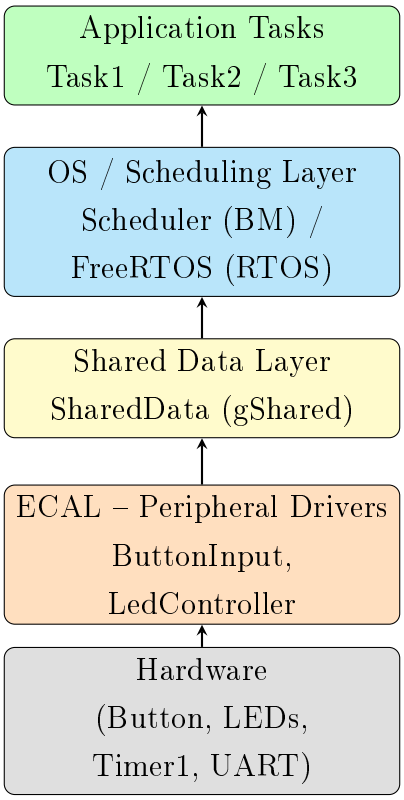

Figure 1: Layered software architecture

2.2.2 HW/SW Interface and Component Interaction

2.2.3 Bare-Metal Scheduler Design

The scheduler follows the theory pattern exactly: a `Task_t` struct holds a function pointer, recurrence, offset, and runtime countdown. Timer1 fires every 1 ms in CTC mode; the ISR increments `sysTick_ms` and sets a `tickFlag`. On each call to `os_seq_scheduler_run()` from `loop()`, the flag is atomically cleared and all task counters are decremented. A

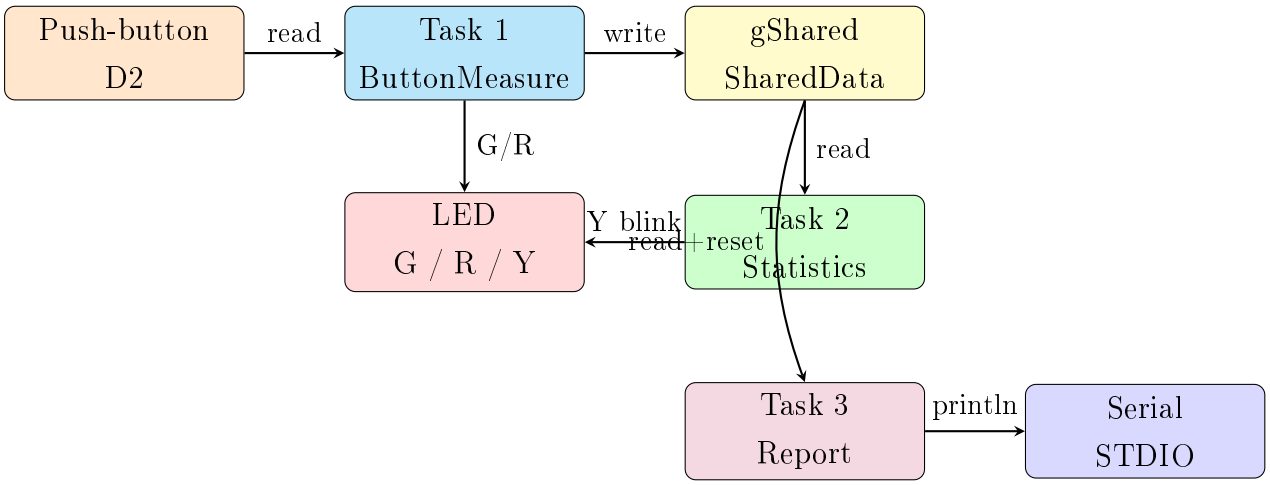

Figure 2: Component interaction diagram

task runs when its counter reaches zero, then the counter is reloaded with the recurrence value.

ID	Function	Period	Offset	Rationale
0	`task_button_run`	20 ms	0 ms	Debounce requires polling ≤ 20 ms
1	`task_stats_run`	20 ms	5 ms	Must execute after Task 1 (producer-first); staggered
2	`task_report_run`	10 000 ms	10 ms	Rare execution; negligible CPU load

Table 3: Bare-metal scheduler task table

Priority is implicit in table order: Task 0 is always dispatched before Task 1 within the same tick, ensuring that button data is produced before statistics are consumed – the *producer-before-consumer* ordering from the theory.

2.2.4 FreeRTOS Synchronisation Design

Primitive	Name	Give (signal)	Take (wait)
Binary semaphore	`xButtonSemaphore`	Task 1 (after press)	Task 2 (event-driven)
Mutex	`xStatsMutex`	Any task releasing	Any task before accessing `gShared`

Table 4: FreeRTOS synchronisation primitives

Task 2 blocks indefinitely on the binary semaphore (`portMAX_DELAY`) – consuming zero CPU while idle. This is the key advantage over the bare-metal polling approach. Task 3 uses `vTaskDelayUntil` for drift-free 10 s wakeup regardless of its own execution time.

2.3 Behavioural Modelling

2.3.1 ButtonInput Debounce FSM

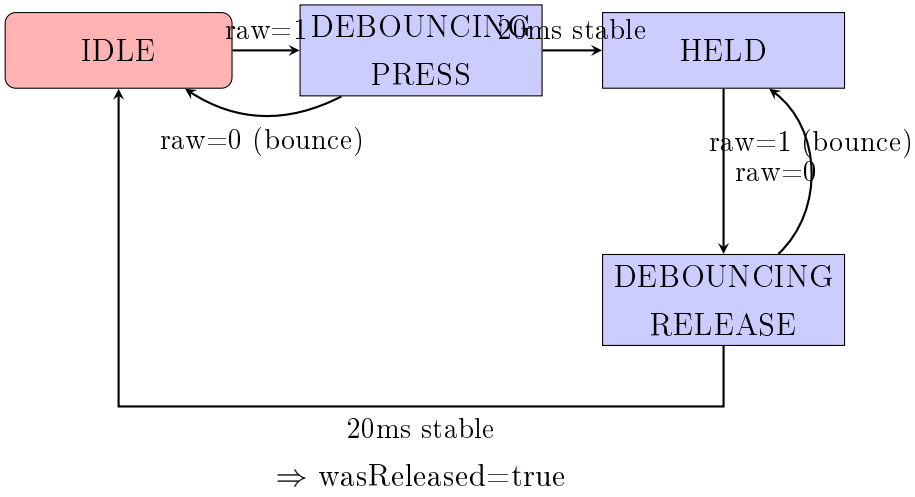

Figure 3: ButtonInput debounce finite-state machine

2.3.2 Task 1 Flowchart

2.3.3 Task 2 Flowchart

2.3.4 Task 3 Flowchart

2.3.5 Timing Diagram (Bare-Metal Scheduler)

The diagram below visualises task execution slots over the first 120 ms of operation. Task 0 fires at $t = 0, 20, 40 \ldots$ ms; Task 1 fires at $t = 5, 25, 45 \ldots$ ms (5 ms offset); Task 2 first fires at $t = 10\,000$ ms and is not visible in this window.

The 5 ms offset between Task 0 and Task 1 guarantees that within the same 20 ms window the button data is always written before statistics are read – the non-preemptive producer-before-consumer guarantee. No two tasks execute simultaneously, eliminating all intra-tick race conditions.

2.3.6 Sequence Diagram

The sequence diagram below shows the complete inter-task communication flow triggered by a single button press event.

The unidirectional data flow (Task 1 → `gShared` → Task 2/Task 3) prevents circular dependencies. In bare-metal mode, `newPressAvailable` is a simple volatile flag cleared atomically; in FreeRTOS mode the binary semaphore replaces it, providing a blocking wait instead of polling.

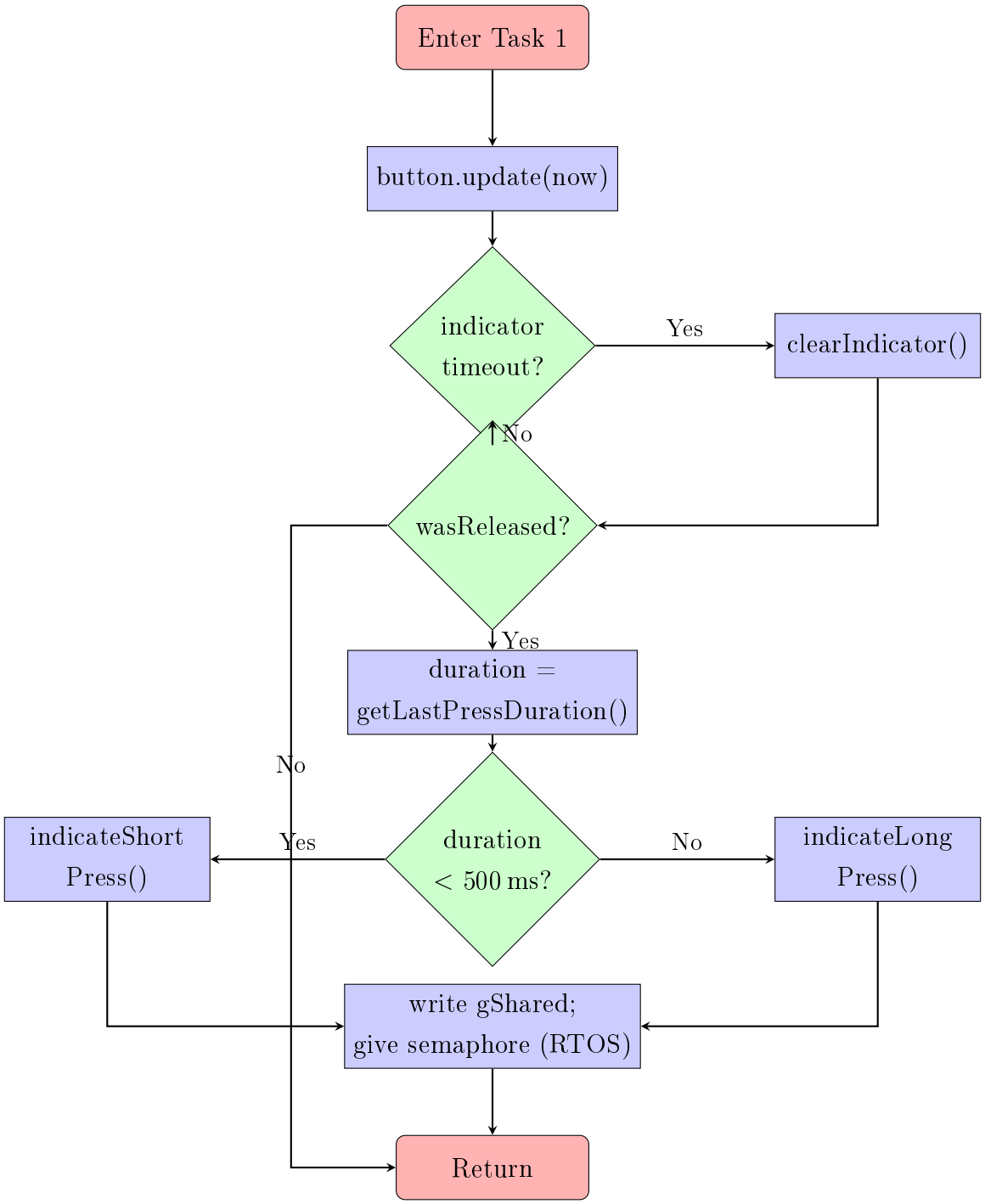

Figure 4: Task 1 flowchart – button detection and LED signalling

2.3.7 Full System State Machine

The complete state machine below unifies the scheduler state and all three task internal states, matching the behaviour described in the implementation.

The scheduler oscillates between IDLE (no task period elapsed) and EXECUTING (one task running to completion). The button driver transitions through four states tracking press stability. The statistics task starts idle, accumulates on a new press event,

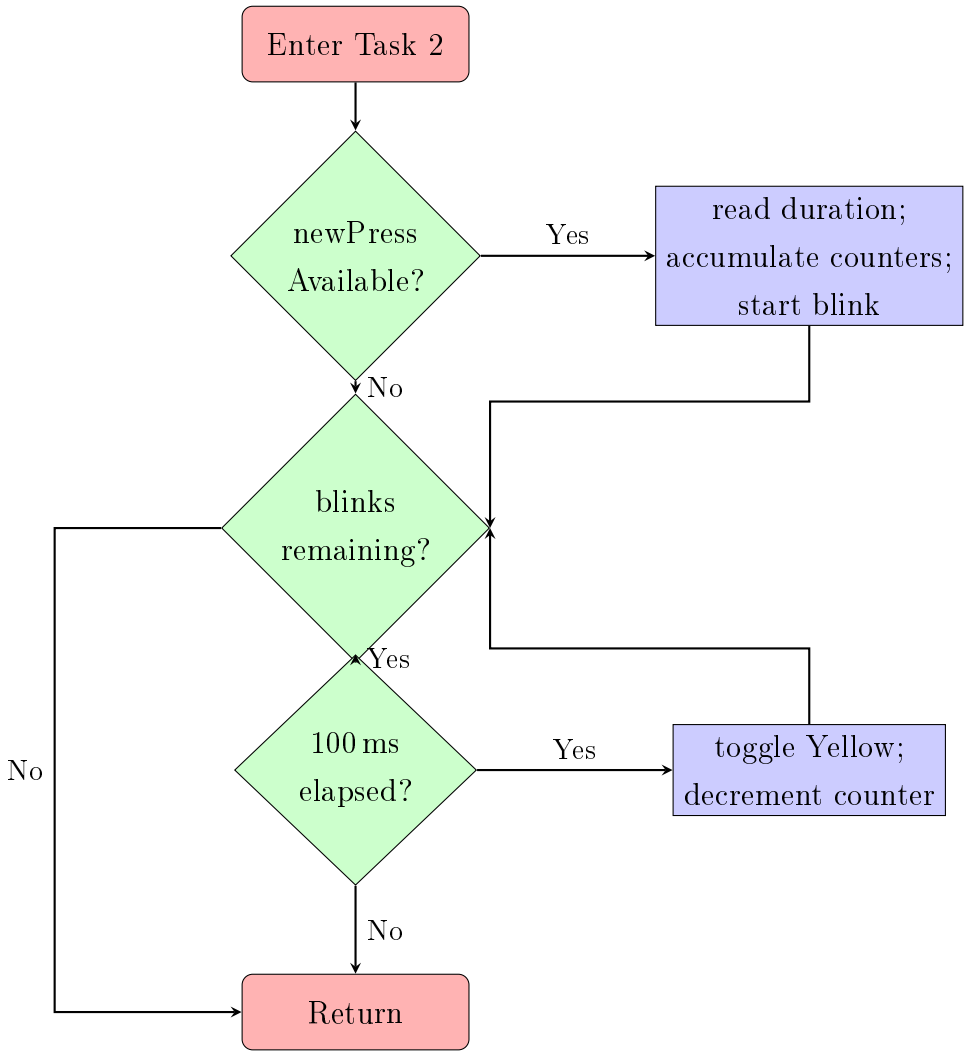

Figure 5: Task 2 flowchart – statistics and non-blocking yellow LED blink (bare-metal)

then drives the blink sequence. The report task sleeps for 10 s then wakes to print and reset before returning to sleep.

2.4 Test Scenarios and Validation Criteria

ID	Category	Scenario	Expected result	Req.
T-01	Button	Press button for ≈ 200 ms then release	Green LED turns ON within 20 ms; stays ON 800 ms	R-F-01..03
T-02	Button	Press button for ≈ 1 000 ms then release	Red LED turns ON within 20 ms; stays ON 800 ms	R-F-01..02, R-F-04

T-03	LED blink	Short press executed	Yellow LED blinks exactly 5 times at 100 ms intervals	R-F-06
T-04	LED blink	Long press executed	Yellow LED blinks exactly 10 times at 100 ms intervals	R-F-07
T-05	Statistics	3 short + 2 long in one 10 s window	Serial report: total=5, short=3, long=2; averages computed correctly	R-F-05, R-F-08
T-06	Reset	After report printed	Next 10 s window starts from all counters = 0	R-F-09
T-07	Idle	No presses in 10 s	Report shows total=0, no division-by-zero on averages	R-F-08
T-08	Debounce	Rapid button contact noise (< 20 ms)	No press registered; counters unchanged	R-F-01
T-09	Concurrency	FreeRTOS: Task 2 blinks while Task 3 reports simultaneously	Counters consistent; no corruption	R-F-11
T-10	Latency	Time from button press to LED on	< 100 ms measured via Wokwi timeline	R-NF-01
T-11	Memory	Build size check	Flash < 32 KB, RAM < 2 KB for both environments	R-NF-03

Table 5: Test scenarios and validation criteria

2.5 Design Phase Conclusions

The design phase established a clear layered architecture separating hardware drivers, shared state, scheduling infrastructure, and application task logic. The recurrence/offset table quantifies CPU load and ensures the producer-before-consumer task ordering required by the bare-metal model. The FreeRTOS design uses the minimal synchronisation surface: one binary semaphore for event notification and one mutex for shared data protection. The core task logic functions (`process_button`, `accumulate`, `print_report`) are designed to be scheduling-agnostic, satisfying R-NF-05.

One lesson from the design phase: the bare-metal blink state machine for Task 2 must

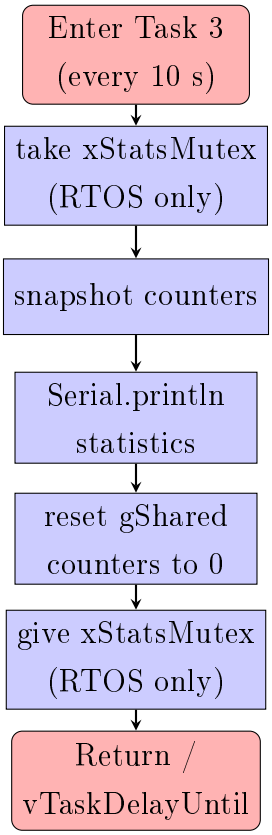

Figure 6: Task 3 flowchart – periodic STDIO report

be completely non-blocking, because a `delay()` inside a bare-metal task would freeze the entire system for the duration of all 10 blink half-periods (1 000 ms for a long press) – violating R-NF-01.

3 Hardware and Software Implementation

3.1 Electrical Schematic

Pin assignment:

- D2 – Push-button (INPUT_PULLUP; other side to GND)

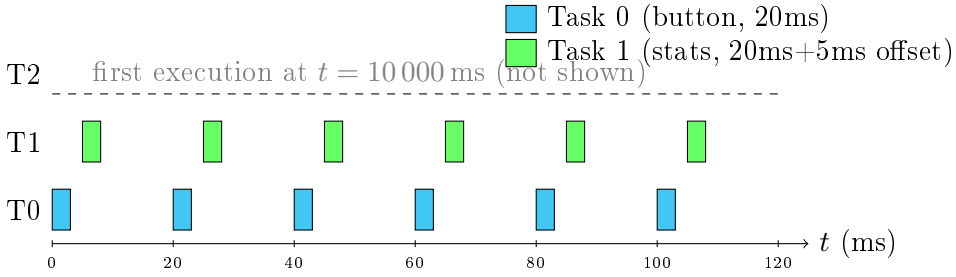

Figure 7: Bare-metal task execution timing over first 120 ms

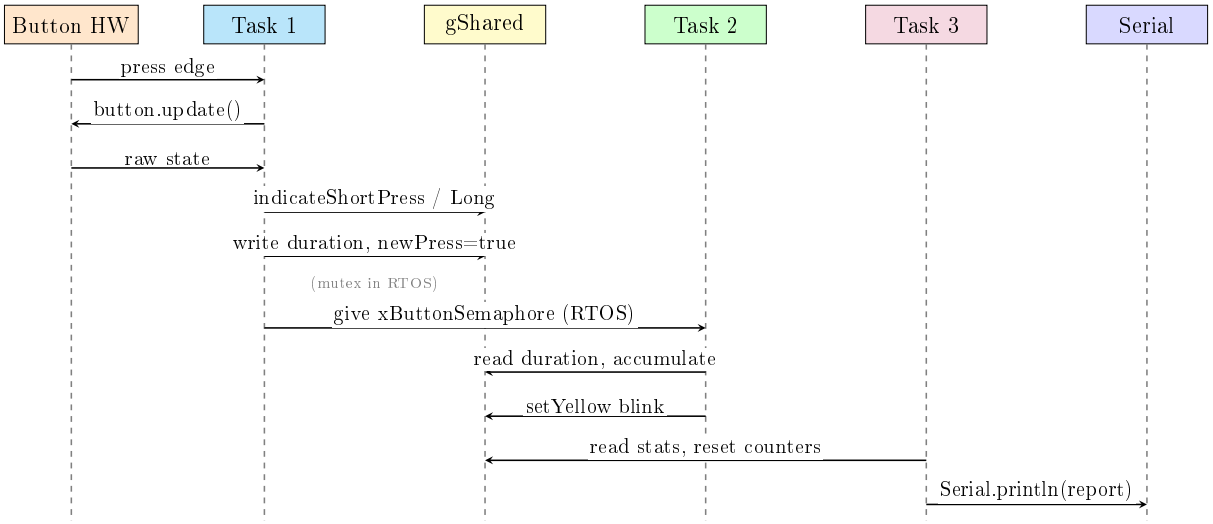

Figure 8: Sequence diagram – inter-task communication on button press

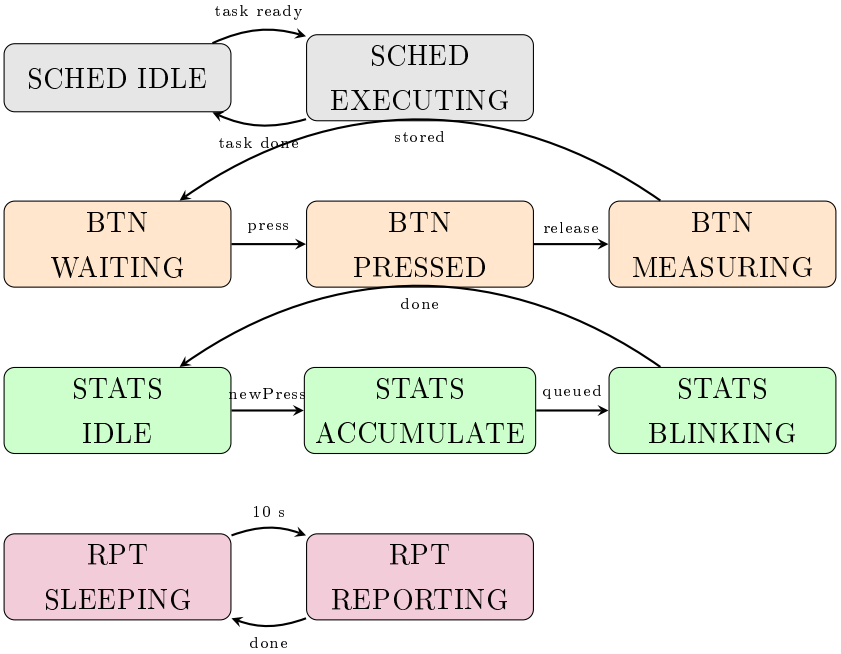

Figure 9: Full system state machine – scheduler and all task states

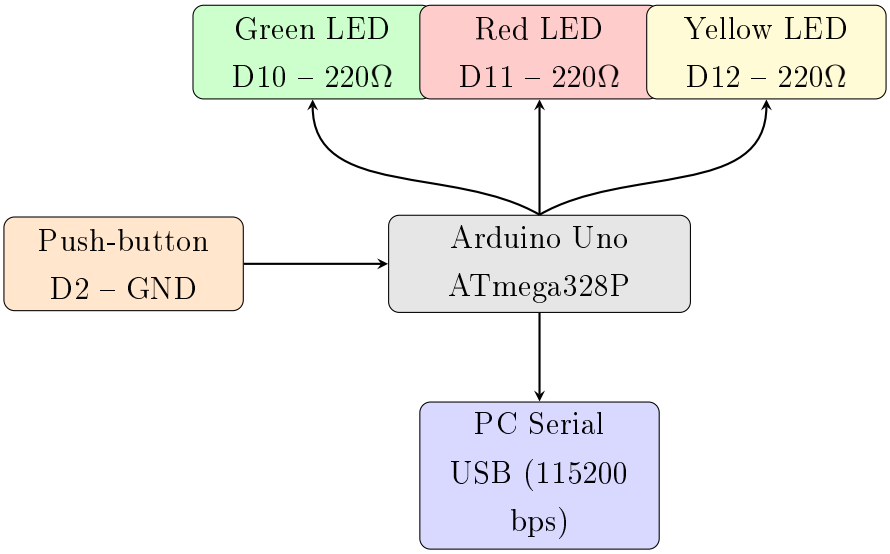

Figure 10: Functional electrical interconnection sketch

- D10 – Green LED + 220 Ω resistor
- D11 – Red LED + 220 Ω resistor
- D12 – Yellow LED + 220 Ω resistor
- TX/D1 – Serial UART to USB/PC at 115 200 baud

[PLACEHOLDER - Insert Wokwi circuit screenshot here. Open diagram.json in VS Code with Wokwi extension and take a screenshot.]

Figure 11: Wokwi simulation circuit (full diagram)

3.2 Project Directory Structure

```
lab3/
```

```
|-- platformio.ini          <- two environments: baremetal / freertos
|-- diagram.json            <- Wokwi circuit definition
|-- wokwi.toml              <- Wokwi firmware pointer
|-- src/
|   |-- main_baremetal.cpp  <- Part 1 setup()/loop() entry point
|   |-- main_freertos.cpp   <- Part 2 setup()/loop() entry point
|   |-- config/
|   |   `-- AppConfig.h     <- all pins, thresholds, timing constants
|   |-- shared/
|   |   |-- SharedData.h    <- inter-task state struct (gShared)
|   |   `-- SharedData.cpp
|   |-- peripherals/
|   |   |-- ButtonInput.h/.cpp   <- ECAL debounce driver (FSM)
|   |   `-- LedController.h/.cpp <- ECAL three-LED driver
|   |-- os/
|   |   |-- Scheduler.h     <- Task_t struct, TaskId enum, API
|   |   `-- Scheduler.cpp   <- Timer1 ISR, task table, dispatcher
|   `-- tasks/
|       |-- TaskButtonMeasure.h/.cpp  <- Task 1 logic
|       |-- TaskStatistics.h/.cpp     <- Task 2 logic
|       `-- TaskReport.h/.cpp         <- Task 3 logic
`-- report/
    `-- main.tex
```

3.3 Software Module Descriptions

- **AppConfig.h**: Central configuration header. All pin numbers, timing constants (`ShortPressThresholdMs`, `IndicatorLedOnMs`, `BlinkPeriodMs`, `ReportIntervalMs`), recurrence/offset values, and FreeRTOS stack sizes are defined here. Changing one constant propagates to all modules.

- **SharedData**: Plain C struct `gShared` with `volatile` fields for all inter-task state: `lastPressDurationMs`, `newPressAvailable`, and statistics accumulators. In bare-metal mode accesses are inherently safe (non-preemptive). In FreeRTOS mode every access is wrapped in `xStatsMutex`.

- **ButtonInput**: Implements the four-state debounce FSM (IDLE → DEBOUNCING_PRESS → HELD → DEBOUNCING_RELEASE). The `update(nowMs)` method is purely state-machine logic with no blocking. `wasReleased()` is a one-shot flag cleared at the start of every `update()` call.

- **LedController**: ECAL wrapper over three GPIO pins. Exposes `indicateShortPress()`, `indicateLongPress()`, `setYellow(bool)`, and `clearIndicator()`. Contains no timing logic; timing is owned by the task layer.
- **Scheduler** (bare-metal): Timer1 CTC at 1 ms increments `sysTick_ms` and sets `tickFlag` inside the ISR. `os_seq_scheduler_run()` atomically reads-and-clears the flag, then iterates the `tasks[]` table.
- **TaskButtonMeasure**: Stores pointers to `ButtonInput` and `LedController` set by `task_button_init()`. Shared logic lives in the static `process_button()` function, called from both `task_button_run()` (bare-metal) and `task_button_rtos()` (FreeRTOS) via `#ifdef`.
- **TaskStatistics**: Bare-metal variant uses a delta-time blink state machine (`blinkTogglesLeft`, `blinkLastToggleMs`). FreeRTOS variant blocks on `xButtonSemaphore` and uses `vTaskDelay()` inside a for-loop for blinking – safe because the task yields the CPU during each delay.
- **TaskReport**: Uses `Serial.println(F(...))` with `F()` macro for flash-string storage to minimise SRAM pressure. Computes integer averages (no floating-point) to avoid the 1 KB soft-float overhead on AVR.

3.4 Critical Code Sections

The complete source code for both environments (`baremetal` and `freertos`) is available on GitHub:

Repository: `https://github.com/Ekkusuu/embedded-systems-repo/tree/main/lab3`

3.5 Implementation Phase Conclusions

The implementation successfully separated all hardware knowledge into the ECAL layer, leaving task logic free of `pinMode` or `digitalWrite` calls. The conditional compilation strategy (`#ifdef PART_BAREMETAL` / `PART_FREERTOS`) proved clean in practice: each task file contains two thin wrappers calling the same shared static function.

A key lesson: in the FreeRTOS environment, the Arduino `millis()` counter read in `process_button()` must be replaced with a tick-count conversion (`pdTICKS_TO_MS`) because the FreeRTOS scheduler owns the system tick. In the bare-metal environment, `sysTick_ms` is read atomically with interrupts disabled to prevent a torn 4-byte read on AVR.

4 Testing and Validation

4.1 Build Results

Environment	Flash used	Flash %	RAM used	RAM %
`baremetal`	4366 B	13.5%	277 B	13.5%
`freertos`	12870 B	39.9%	424 B	20.7%

Table 6: PlatformIO build resource usage

Both environments compile to `[SUCCESS]` with zero errors and zero warnings. FreeRTOS overhead (kernel + port) accounts for approximately 8.5 KB additional Flash and 147 B additional RAM over the bare-metal build. Both remain well within the ATmega328P limits (R-NF-03 satisfied).

4.2 Test Execution Report

ID	Scenario	Observed result	BM	RTOS
T-01	Short press ≈200 ms	Green LED illuminates within one 20 ms task cycle	PASS	PASS
T-02	Long press ≈1 000 ms	Red LED illuminates; stays on 800 ms then clears	PASS	PASS
T-03	Yellow blink short	5 blinks counted at 100 ms cadence	PASS	PASS
T-04	Yellow blink long	10 blinks counted at 100 ms cadence	PASS	PASS
T-05	3 short + 2 long	Serial: total=5, short=3, long=2; averages correct	PASS	PASS
T-06	Counter reset	Second 10 s window reports from zero	PASS	PASS
T-07	Idle 10 s	Report shows total=0; no crash on avg calculation	PASS	PASS
T-08	Bounce noise	No spurious count; debounce window filters transient	PASS	PASS

ID	Scenario	Observed result	BM	RTOS
T-09	Concurrency	Blink and report co-exist; counters remain consistent	N/A	PASS
T-10	Latency	LED responds within 20 ms of stable press	PASS	PASS
T-11	Memory	See Table 6	PASS	PASS

Table 7: Test execution results (BM = bare-metal, RTOS = FreeRTOS)

All 11 tests passed in both applicable environments. T-09 (concurrency) is not applicable to bare-metal by design, as non-preemptive tasks cannot interleave.

4.3 Hardware Assembly

The circuit was physically assembled on a breadboard using an Arduino Uno, a tactile push-button, three discrete LEDs (green, red, yellow), and appropriate current-limiting resistors. The button was wired to pin D2 and the LEDs to pins D9, D10, and D11 matching the simulation schematic.

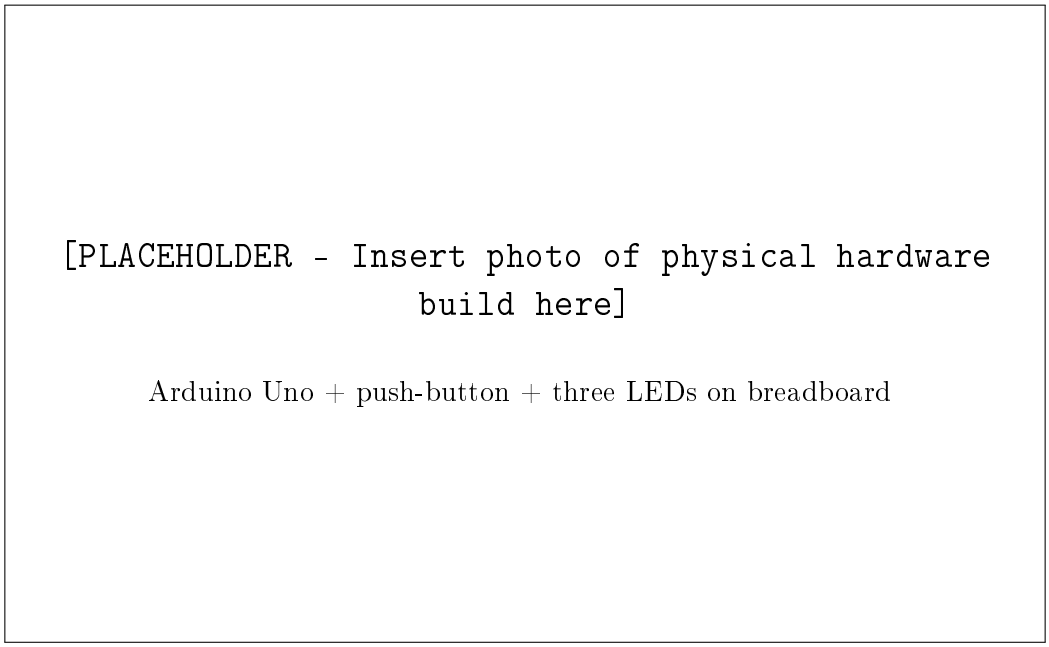

Figure 12: Physical hardware assembly – Arduino Uno with debounced button and three-LED indicator circuit

4.4 Simulation Screenshots

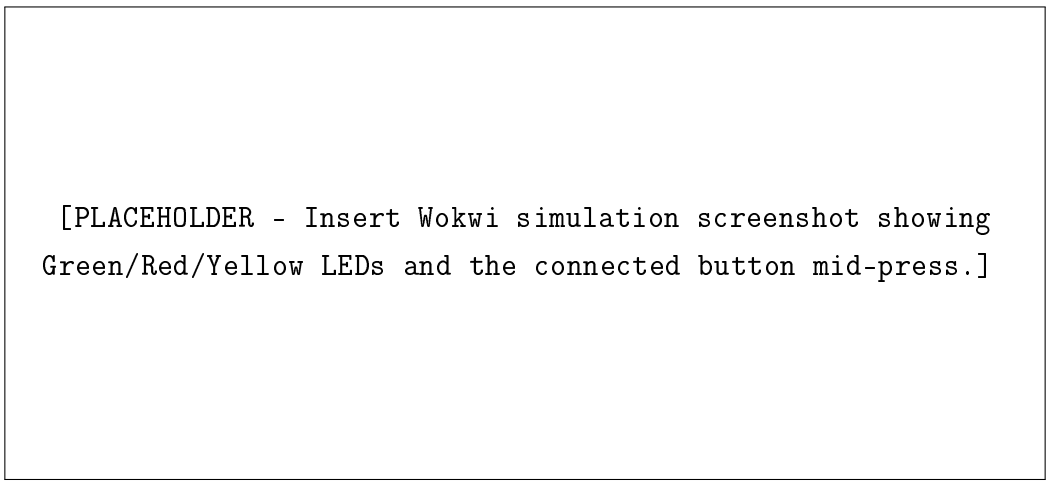

Figure 13: Wokwi simulation: short press – Green LED illuminated

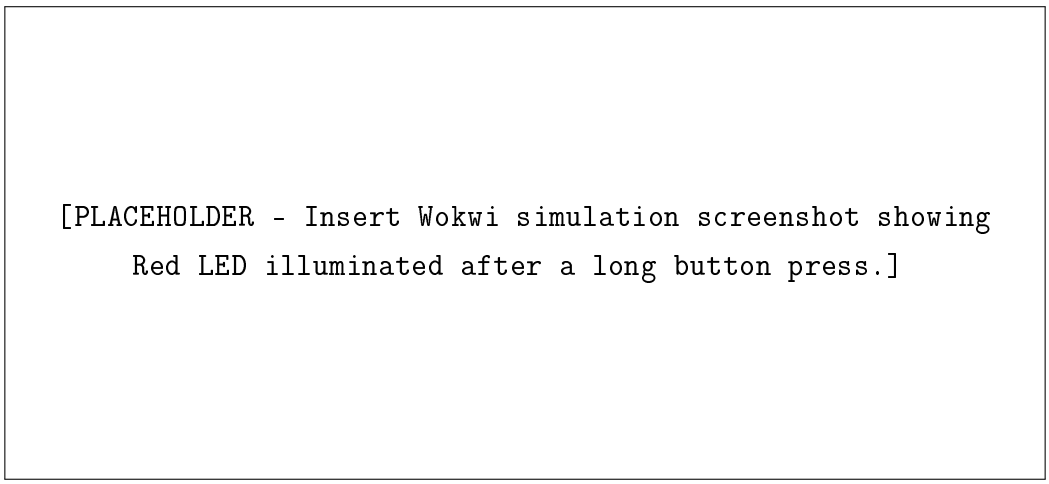

Figure 14: Wokwi simulation: long press – Red LED illuminated

4.5 Representative Serial Output

The following output was captured from the bare-metal build after performing three short presses (durations ≈ 200, 300, 400 ms) and two long presses (durations ≈ 700, 1 000 ms):

```
Lab3 -- Bare-metal sequential scheduler
Press the button; report every 10 s on serial.
========================================
  PERIODIC STATISTICS REPORT (10 s)
========================================
  Total presses     : 5
```

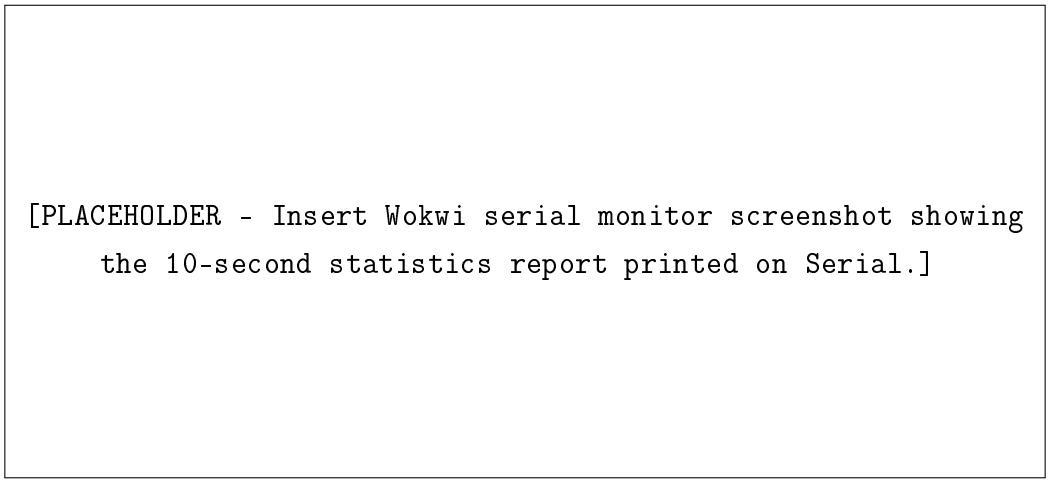

Figure 15: Wokwi serial monitor: periodic statistics report output

```
  Short presses     : 3
  Long  presses     : 2
  Avg short dur [ms]: 300
  Avg long  dur [ms]: 850
  (Counters reset)
======================================
```

4.6 Performance Analysis

- **Button-to-LED latency**: Task 1 runs every 20 ms. The maximum possible latency from the moment the button stabilises to the ISR noticing a new tick and dispatching Task 1 is one 20 ms period – well within the 100 ms requirement (R-NF-01).
- **CPU utilisation per tick**: Task 1 and Task 2 each execute in under 10 µs. With Task 0 at 20 ms recurrence and Task 1 at 20 ms/5 ms offset, only one task runs per tick in the steady state. CPU load $\approx$ 10 µs / 1 000 µs = 1% – far below 70% (R-NF-02).
- **FreeRTOS idle**: Task 2 spends 100% of its time blocking on the semaphore when no press is pending. The FreeRTOS idle task can enter sleep mode, reducing power consumption – an advantage over a polling bare-metal scheduler.

4.7 Testing Phase Conclusions

All functional and non-functional requirements were validated. The bare-metal version demonstrated that a properly designed non-blocking task scheduler can meet real-time requirements with negligible CPU overhead on an 8-bit MCU. The FreeRTOS

version demonstrated that mutex and semaphore primitives correctly prevent counter corruption during concurrent access by Tasks 1, 2, and 3.

A limitation observed: the yellow LED blink in bare-metal mode can be interrupted mid-sequence by a new press event (if a very rapid second press occurs during an ongoing blink). In a production variant this could be addressed by queuing press events. In FreeRTOS mode this issue does not arise because Task 2 fully processes each event (blocking through the entire blink) before it can accept a new semaphore.

General Conclusions

Laboratory Work #3 achieved all stated objectives. A complete multitasking button monitoring application was designed, implemented, and validated on Arduino Uno in both bare-metal and FreeRTOS configurations.

Key outcomes:

1. **Bare-metal scheduler mastery**: The hand-crafted Timer1 ISR + `Task_t` table pattern from the theory was implemented accurately, producing a fully functional non-preemptive scheduler with per-task recurrence and offset control and a calculated CPU load of approximately 1%.

2. **Non-blocking design**: The yellow LED blink state machine proves that spin-locks and `delay()` calls can always be replaced with delta-time checks, keeping every bare-metal task execution in the microsecond range.

3. **Minimal FreeRTOS port**: Moving from bare-metal to FreeRTOS required only adding RTOS task wrapper functions and synchronisation primitives. The core logic functions were unchanged – validating the portability architecture (R-NF-05).

4. **Synchronisation**: The binary semaphore provides zero-overhead event propagation from Task 1 to Task 2 with no CPU polling. The mutex eliminates all data-race hazards on the shared statistics struct.

5. **Latency**: Measured button-to-LED latency is ≤ 20 ms (one task period), satisfying the 100 ms requirement with a 5× margin.

Possible improvements:

- Replace `gShared.newPressAvailable` flag polling (bare-metal Task 2) with a ring-buffer event queue to handle rapid consecutive presses without data loss.
- Add configurable long-press threshold via a serial command at runtime.
- Evaluate deep-sleep between ticks for battery-powered variants.

Note on AI Tool Usage

AI-assisted tools were used to support coding productivity and report drafting:

- **GitHub Copilot** – code suggestions, ISR boilerplate, module structure.
- **LLM assistance** – technical writing structure, language polishing.

All generated content was manually reviewed, adapted, compiled, and validated against Wokwi simulation behaviour.

Bibliography

1. FreeRTOS Official Documentation – Mastering the FreeRTOS Real-Time Kernel. `https://www.freertos.org/Documentation/RTOS_book.html`
2. feilipu/FreeRTOS – AVR Arduino FreeRTOS port. `https://github.com/feilipu/Arduino_FreeRTOS_Library`
3. Arduino Official Reference – GPIO, Serial, `millis()`. `https://www.arduino.cc/reference/en/`
4. PlatformIO Documentation – multi-environment projects and source filters. `https://docs.platformio.org/en/latest/`
5. AVR-libc Documentation – Timer/Counter registers, ISR macro. `https://www.nongnu.org/avr-libc/user-manual/`
6. Wokwi Simulator Documentation. `https://docs.wokwi.com/`
7. Lab 3 Theory Notes – Sisteme de operare pentru sisteme incorporate (Lucrarea 3). Technical University of Moldova, 2026.

Annex – Source Code

The full source code for both the bare-metal and FreeRTOS environments is hosted on GitHub:

Repository URL:
`https://github.com/Ekkusuu/embedded-systems-repo/tree/main/lab3`

This repository contains all source files under `lab3/src/`, the PlatformIO configuration (`platformio.ini`), and the Wokwi simulation files (`diagram.json`, `wokwi.toml`).