MINISTRY OF EDUCATION AND RESEARCH
OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software and Automation Engineering

VREMERE ADRIAN, FAF-232

Report

Laboratory Work n.2.1

Button Press Duration Monitoring with Non-Preemptive Task Scheduler

on Embedded Systems

Checked by:
Martîniuc Alexei, *univ. asist.*
FCIM, UTM

Chişinău – 2026

1. Domain Analysis

1.1. Objective of the Laboratory Work

The objective of this laboratory work is to design and implement a multitasking bare-metal application on an Arduino Mega 2560 microcontroller that monitors the duration of button presses, provides visual feedback through coloured LEDs, and periodically reports statistics to the user via the STDIO serial interface. The application demonstrates non-preemptive cooperative scheduling — a foundational technique used in resource-constrained embedded systems where no real-time operating system (RTOS) is available.

The key learning objectives include understanding non-preemptive task scheduling with period- and offset-based dispatch, implementing input debouncing and duration measurement using a finite state machine, structuring a multitasking embedded application using task context structures and recurrence arrays, and integrating multiple cooperative tasks that communicate through shared global state without locks or semaphores.

1.2. Problem Definition

The laboratory requires the development of a microcontroller-based application structured as a set of non-preemptive tasks with the following functional requirements:

1. Implement a non-preemptive bare-metal scheduler using task context structures, each containing a function pointer, a recurrence period, and a startup offset. Only one task must be active per scheduler tick.
2. Task 1 — Button Detection and Duration Measurement: monitor the state of a push button, detect press/release transitions using a debounce state machine, measure press duration in milliseconds, and signal the result visually by lighting a green LED for short presses ($< 500\,\text{ms}$) or a red LED for long presses ($\geq 500\,\text{ms}$).
3. Task 2 — Statistics Accumulation and Blinking: at each detected press, increment press counters and duration accumulators, and drive a rapid yellow LED blink sequence: 5 blinks for a short press and 10 blinks for a long press.
4. Task 3 — Periodic STDIO Reporting: every 10 seconds, transmit a formatted summary (total presses, short/long counts, average duration) over the serial STDIO interface, then reset all accumulators for the next measurement window.

1.3. Used Technologies

GPIO and Digital Input/Output

General-Purpose Input/Output (GPIO) pins are the fundamental building blocks for interfacing a microcontroller with external digital components. On the Arduino Mega 2560 (ATmega2560), GPIO pins are configured via the `pinMode()` function and read/written with `digitalRead()`

and `digitalWrite()`. For a push button, the pin is configured as `INPUT_PULLUP`: the internal pull-up resistor holds the line HIGH when the button is open, and the button connects the line to GND when pressed, producing a LOW reading. This eliminates the need for an external pull-up resistor and ensures a defined signal level in both button states.

Button Debouncing

Mechanical push buttons suffer from contact bounce: when a button is pressed or released, the contacts briefly oscillate between open and closed states before settling. This typically lasts for 5–50 ms and produces multiple spurious edge transitions that the microcontroller would otherwise interpret as rapid consecutive presses. Software debouncing addresses this by requiring that the new button state persist for a minimum confirmation period (typically 30–50 ms) before the state change is accepted. In this application, a four-state FSM (Idle, Debounce-Down, Pressed, Debounce-Up) implements debouncing on both the press and release edges, using a 50 ms confirmation window.

Non-Preemptive Cooperative Scheduling (Bare-Metal)

Non-preemptive scheduling, also known as cooperative multitasking, is a software design pattern for bare-metal embedded systems that require apparently concurrent task execution without an RTOS. Each task is implemented as a short, non-blocking function. A central scheduler loop iterates an array of task descriptors on every system tick and dispatches at most one due task per tick. A task descriptor contains three user-defined fields: a function pointer, a recurrence period (in milliseconds), and a startup offset that staggers initial execution to avoid all tasks firing simultaneously.

Compared to RTOS-based preemptive scheduling, the cooperative approach eliminates context-switching overhead, stack-per-task memory cost, and complex synchronisation primitives. Its primary constraint is that every task function must return quickly; a blocking call in one task delays all others. This constraint is acceptable in the current application because all tasks are small and bounded in duration.

The scheduler achieves its timing discipline by maintaining an absolute `nextRun` timestamp for each task, computed as `nextRun += period` after each execution. This avoids cumulative drift that would result from computing `nextRun = millis() + period` at the end of each run. This technique is known as deadline-relative advancement and is standard in professional bare-metal schedulers.

UART Serial Communication and STDIO

The ATmega2560 UART0 is connected through the on-board USB-to-Serial bridge to the host PC. By redirecting the AVR libc standard streams (`stdout`, `stdin`) to the serial hardware, standard C functions `printf()` and `fgets()` send and receive data over the USB connection. This

STDIO redirection technique is implemented in the reusable `StdioSerial` library introduced in Laboratory Work 1.1.

1.4. Hardware Components

Arduino Mega 2560

The Arduino Mega 2560 is a microcontroller development board based on the ATmega2560, an 8-bit AVR microcontroller running at 16 MHz. It provides 54 digital I/O pins, 16 analogue inputs, 256 KB Flash, 8 KB SRAM, and 4096 bytes of EEPROM. Four hardware UART interfaces are available. For this laboratory, four GPIO pins are used: one for button input and three for LED outputs.

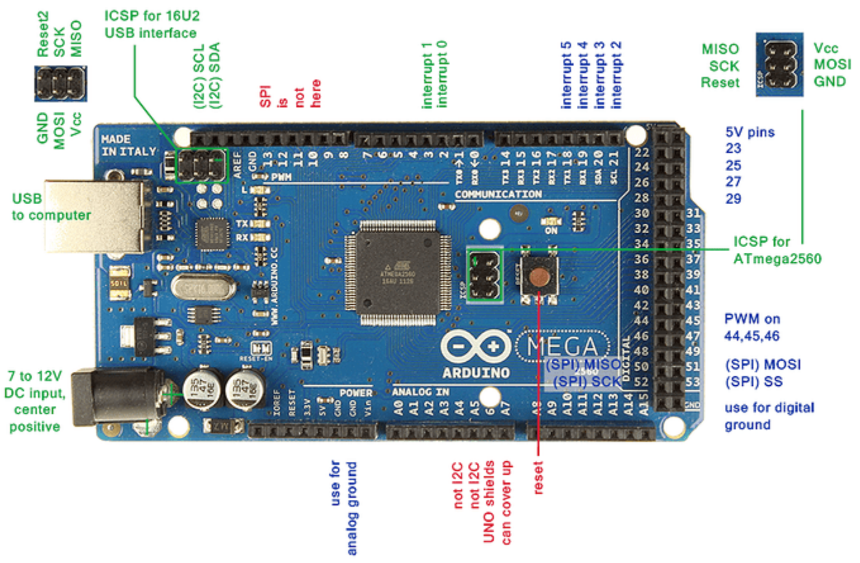


Figure 1.1 *– Arduino Mega 2560 development board*

Push Button

The circuit uses a standard momentary-action tactile push button. When pressed, it connects pin 7 of the microcontroller to GND, making the `digitalRead()` return `LOW`. The ATmega2560 internal pull-up is enabled, so no external resistor is required on this line. Button bouncing is handled in software (see Debouncing above). The button is the sole input device for this application.

Figure 1.2 *– Momentary tactile push button*

LEDs and Current-Limiting Resistors

Three 5 mm LEDs — green, red, and yellow — provide the visual output of the system. Each LED requires a forward voltage of approximately 2.0 V (for red/yellow) or 2.2 V (for green) and a safe forward current of 10–20 mA. To limit the current from the 5 V GPIO pin, a 220 Ω resistor is placed in series with each LED, resulting in a drive current of approximately 12–14 mA, which is within the rated maximum for both the LEDs and the ATmega2560 GPIO pins (40 mA absolute maximum per pin).

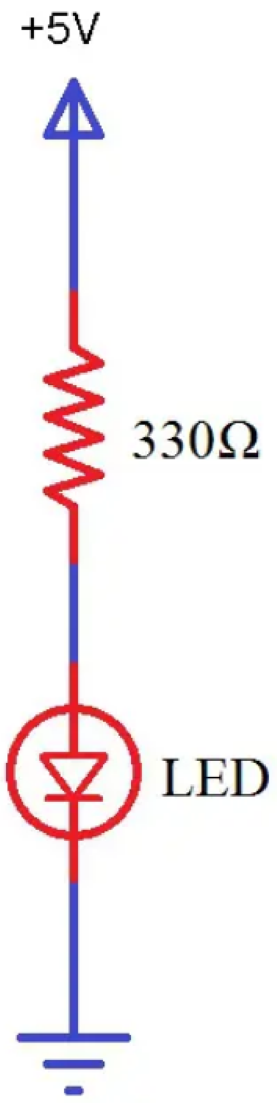
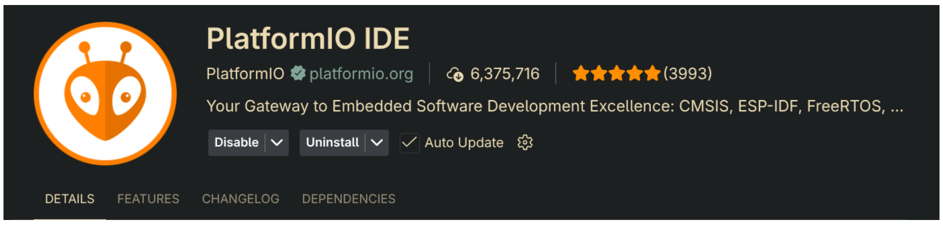


Figure 1.3 *– Standard 5 mm LED indicator component used for visual signaling*

1.5. Software Components

PlatformIO Build System

PlatformIO is an open-source embedded development ecosystem integrated into VS Code. It handles toolchain management, library resolution, and build configuration. Lab 2.1 is defined as an independent PlatformIO environment (`lab2_1`) in `platformio.ini`, targeting the `megaatmega2560` board with the AVR toolchain. The `build_src_filter` directive ensures only the correct lab entry-point files are compiled into each environment, preserving full isolation between labs.



Your Gateway to Embedded Software Development Excellence: CMSIS, ESP-IDF, FreeRTOS, ...

Figure 1.4 *– PlatformIO workspace in VS Code showing the Lab 2.1 build environment*

Wokwi Simulator

Wokwi is a browser and VS Code-integrated simulator supporting Arduino Mega 2560, ESP32, and other platforms. Each lab has a dedicated Wokwi configuration folder (`labs/wokwi/lab2.1/`) containing a `diagram.json` with the circuit description and a `wokwi.toml` pointing to the compiled firmware binary. The simulator enables functional testing of the button press detection, LED signaling, and serial report output without physical hardware.

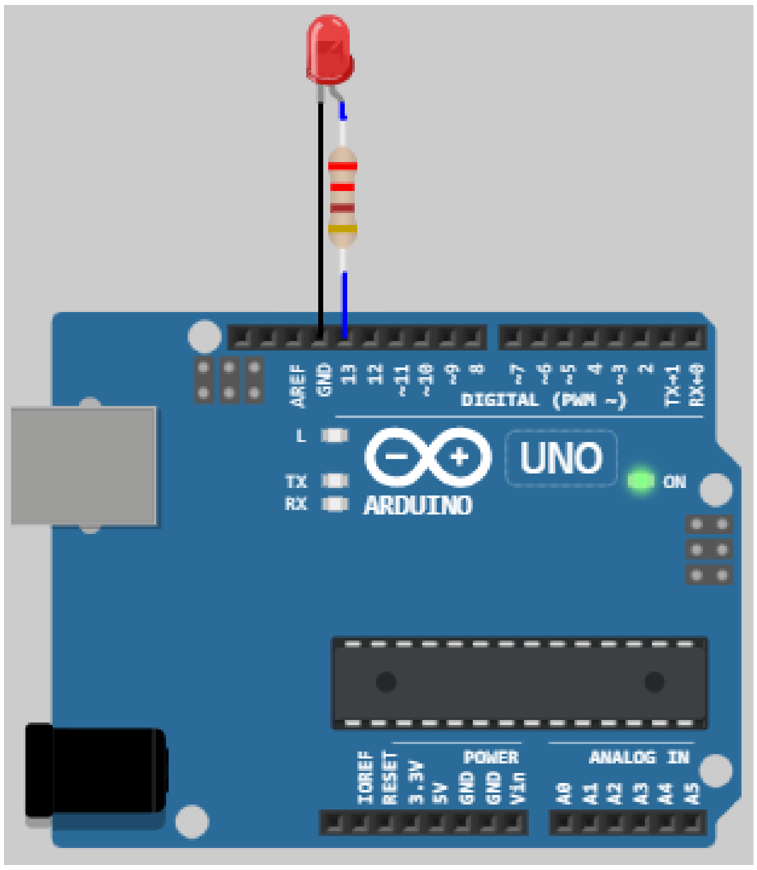


Figure 1.5 *– Wokwi VS Code plugin used for circuit simulation and firmware testing*

TaskScheduler Library

The `TaskScheduler` library is a new reusable module created specifically for this laboratory. It provides a minimal non-preemptive cooperative scheduler based on an array of `TaskContext_t` structures. Each structure stores a function pointer, a period, an offset, and a computed `nextRun` timestamp. The `schedulerRun()` function selects the most-overdue due task on each call and dispatches it exactly once, making it straightforward to express each application concern as an independent, testable, non-blocking function.

StdioSerial Library

The `StdioSerial` library, originally developed in Laboratory Work 1.1, redirects the AVR libc standard I/O streams to UART0 for use with `printf()` and `fgets()`. It is used without modification in Lab 2.1 to output the periodic statistics report from Task 3.

1.6. Case Study: Industrial Process Monitoring

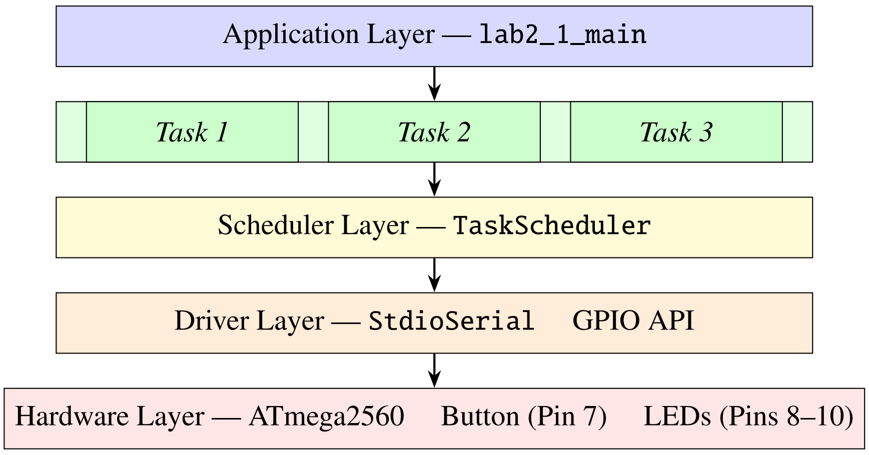
The combination of periodic sampling, event detection, and timed reporting demonstrated in this lab is directly applicable to industrial condition monitoring. In manufacturing environments, vibration sensors or limit switches monitor machine operating cycles. An embedded controller counts actuations, classifies them by duration or force, and transmits summary statistics to a SCADA system at regular intervals. Short cycle times might indicate a correctly operating machine, while anomalously long or short presses may indicate faults requiring maintenance. Non-preemptive bare-metal scheduling is preferred in such safety-relevant applications because its timing is fully deterministic and auditable without relying on an RTOS certification layer.

2. Design

2.1. System Architecture Diagrams

Layered Software Architecture

The software is organised into five layers, each with a clearly bounded responsibility. The Application Layer contains the lab entry point `lab2_1_main`, which initialises all hardware and registers the tasks. The Task Layer holds the three cooperative task functions (Task 1, Task 2, Task 3) that implement the application logic. The Scheduler Layer is the `TaskScheduler` library, which drives periodic dispatch of tasks without preemption. The Driver Layer consists of `StdioSerial` (UART STDIO redirection) and the Arduino GPIO API (`pinMode`, `digitalRead`, `digitalWrite`). At the bottom, the Hardware Layer comprises the ATmega2560 MCU, the push button, and the three indicator LEDs.



Application Layer — lab2_1_main
Task 1
Task 2
Task 3
Scheduler Layer — TaskScheduler
Driver Layer — StdioSerial GPIO API
Hardware Layer — ATmega2560 Button (Pin 7) LEDs (Pins 8–10)

Figure 2.1 *– Layered software architecture of the Lab 2.1 application*

System Structural Diagram

The system involves a host PC communicating bidirectionally with the microcontroller over USB (through UART0) and four physical peripherals connected to GPIO pins. The button is the sole input, while three LEDs serve as outputs. The TaskScheduler orchestrates all application logic, and Task 3 produces the only serial output (the periodic statistics report).

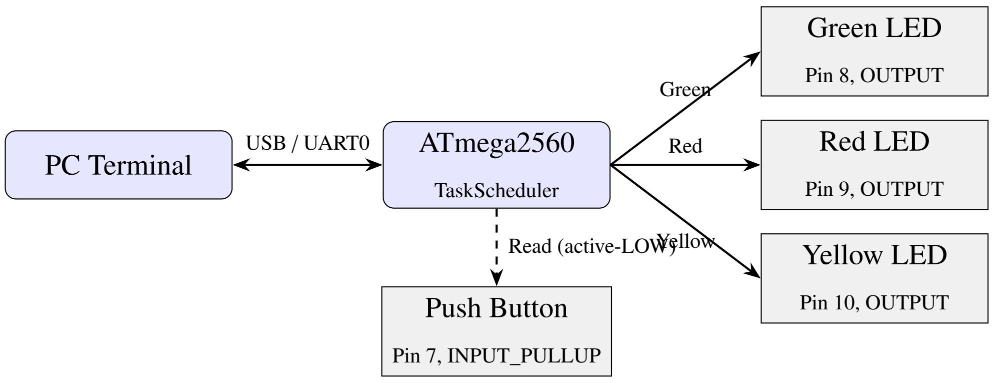


Figure 2.2 *– System structural diagram showing MCU, peripherals, and serial interface*

2.2. Behaviour Diagrams

Non-Preemptive Task Scheduler — Tick Flowchart

The `schedulerRun()` function implements a single scheduling tick. It scans the task array for due tasks, selects the one with the earliest missed deadline (most urgently overdue), runs its function pointer exactly once, and advances the task's next scheduled time. If no task is due, the function returns immediately, allowing the `loop()` to spin rapidly until the next deadline is reached.

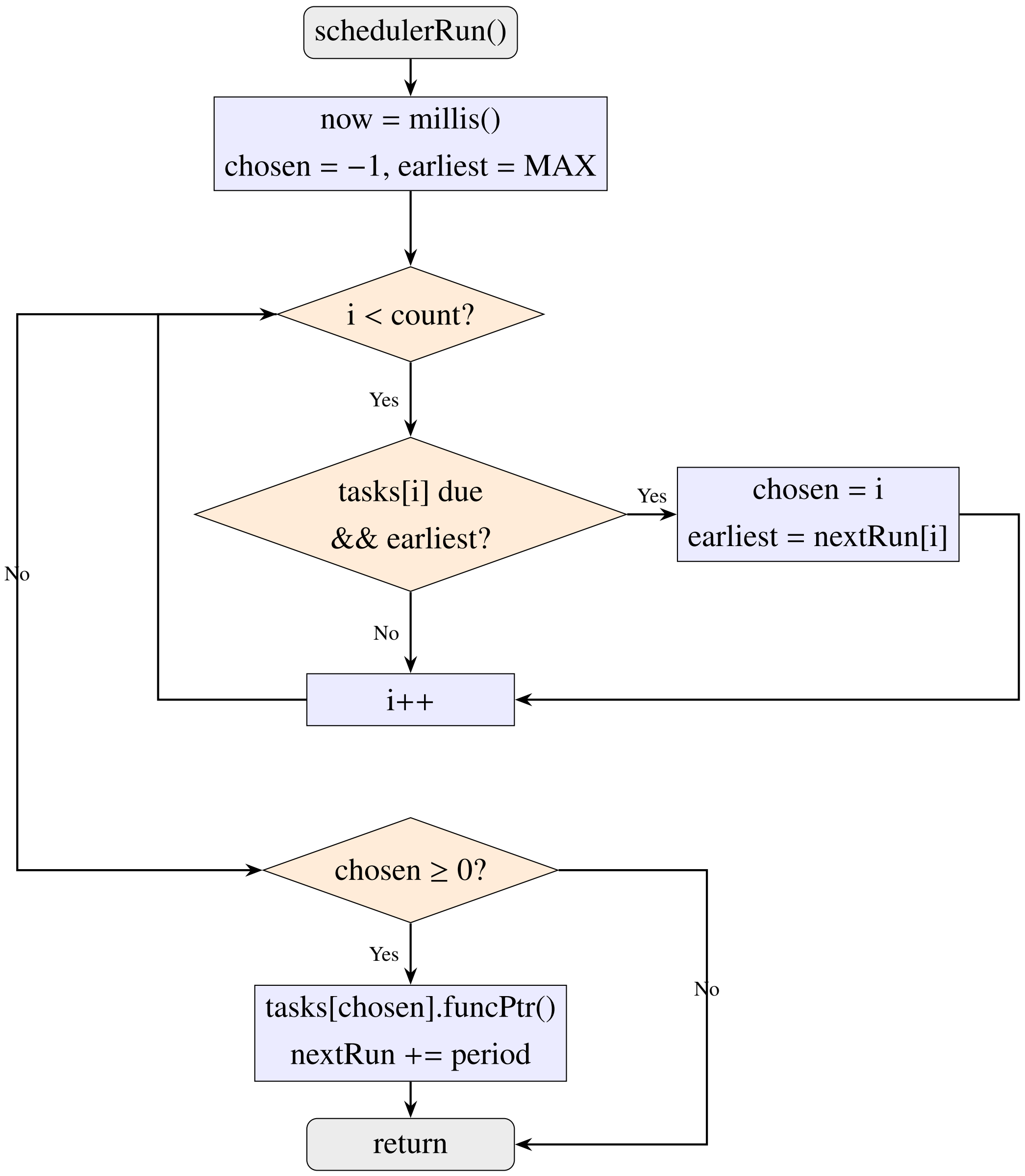


***Figure 2.3** – Flowchart of `schedulerRun()` — one task dispatched per tick*

Button FSM — State Transition Diagram

Task 1 uses a four-state FSM to reliably detect button presses and measure their duration. The debounce confirmation window is 50 ms on both the press and release edges, eliminating contact bounce artefacts. The press start time is recorded on entry to PRESSED state, and the release time is captured when the first HIGH reading occurs during PRESSED. The duration is computed only when the DEBOUNCE_UP state confirms the release.

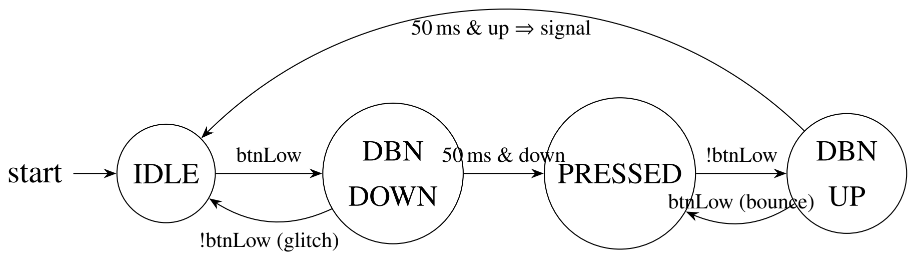


***Figure 2.4** – Button debounce finite state machine with 50 ms confirmation windows*

Task 1 — Button Detection and LED Signaling Flowchart

Task 1 runs every 10 ms. On each tick it reads the button pin, advances the FSM (see *Figure 2.4*), and checks the two LED auto-off timers. When the FSM completes a press cycle, it lights the green LED (short press) or the red LED (long press) for 1 500 ms and signals Task 2 via the `g_newPress` flag.

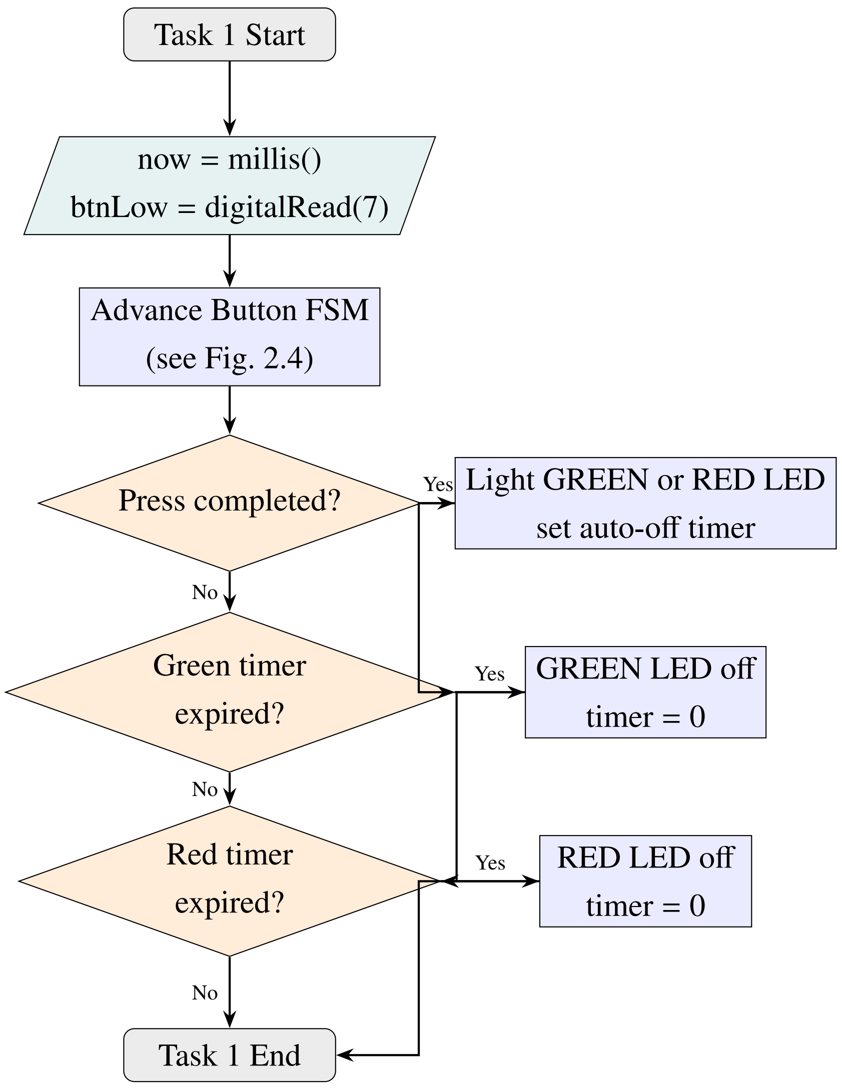


***Figure 2.5** – Task 1 flowchart — button reading, FSM dispatch, and LED timer management*

Task 2 — Statistics and Yellow LED Blink Flowchart

Task 2 runs every 50 ms. It first checks whether Task 1 has signalled a new completed press event. If so, it increments the appropriate counters, arms the yellow LED blink sequencer (10 half-cycles for a short press, 20 for a long press), and clears the event flag. Independently of new events, the blink sequencer is advanced on each run: when 100 ms has elapsed since the last toggle, the yellow LED changes state until all half-cycles are consumed.

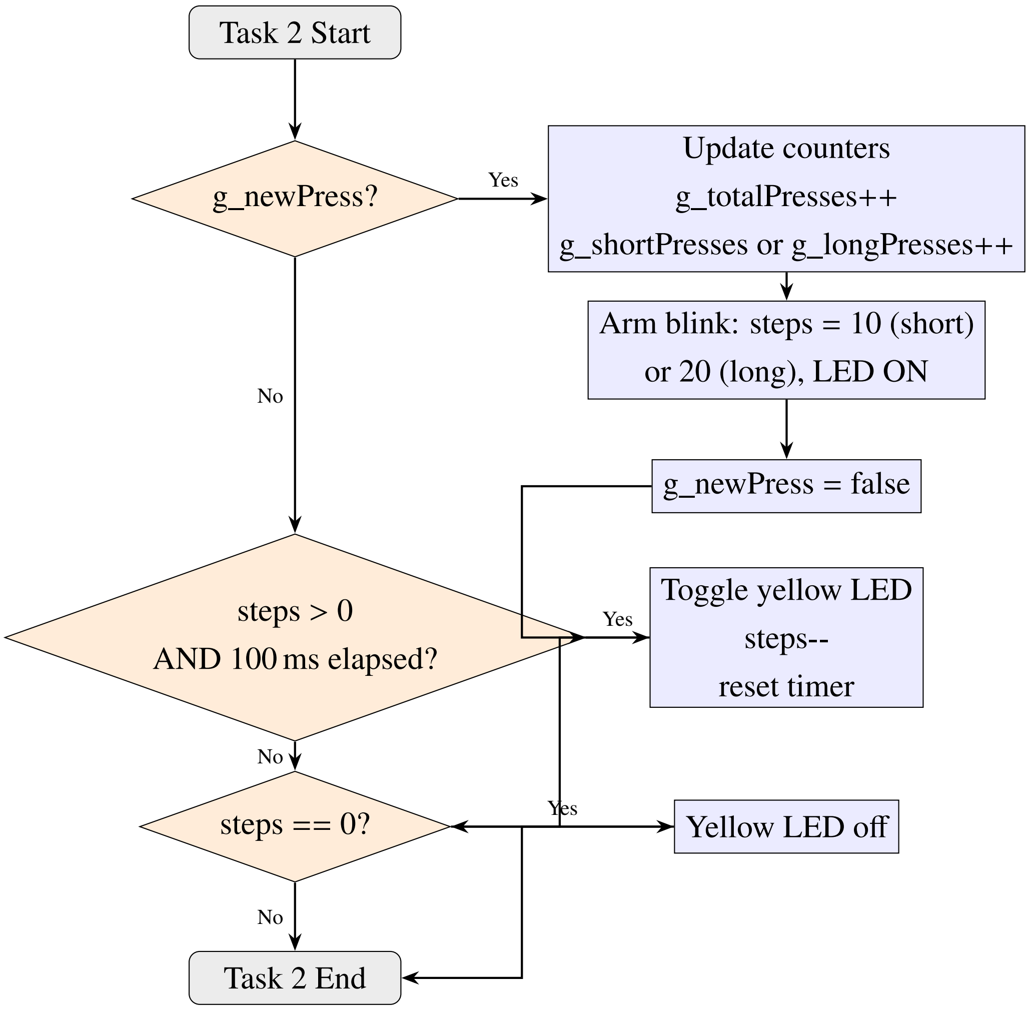


***Figure 2.6** – Task 2 flowchart — statistics update and yellow LED blink sequencer*

Task 3 — Periodic STDIO Report Flowchart

Task 3 runs every 10 000 ms. It snapshots the current accumulator values, computes the average duration, prints a formatted report via `printf()`, and resets all counters. The reset ensures each reporting window is independent, so the user sees the activity in the most recent 10-second window rather than a cumulative total.

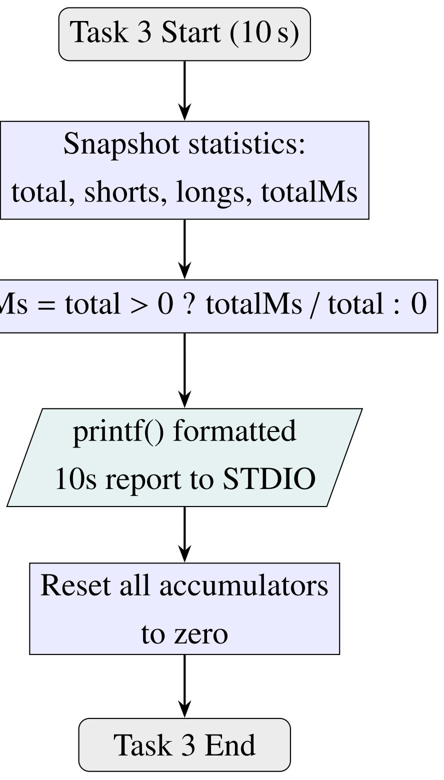

Task 3 Start (10 s)
Snapshot statistics: total, shorts, longs, totalMs
avgMs = total > 0 ? totalMs / total : 0
printf() formatted 10s report to STDIO
Reset all accumulators to zero
Task 3 End

***Figure 2.7** – Task 3 flowchart — periodic statistics report and accumulator reset*

2.3. Electrical Schematics

The circuit comprises four connections to the Arduino Mega 2560: one active-LOW button input on pin 7 (with internal pull-up, no external resistor required), and three LED output circuits on pins 8, 9, and 10. Each LED branch consists of a 220 Ω current-limiting resistor in series with the LED, with the cathode returning to GND.

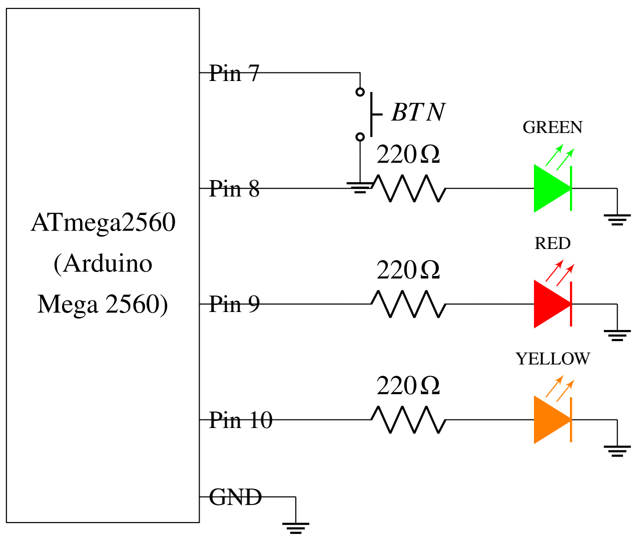


***Figure 2.8** – Electrical schematic — button (active-LOW) and three LED indicator circuits*

Component Specification

Three circuit elements are used per LED branch: the 5 V digital output GPIO pin, a 220 Ω resistor, and a standard 5 mm LED with approximately 2.0 V forward voltage. The resulting current is $I = (5.0 - 2.0)/220 \approx 13.6\,\text{mA}$, well within the ATmega2560 absolute-maximum GPIO sink/source current of 40 mA. The push button requires no external component because the ATmega2560 internal pull-up holds the line at approximately 5 V when the button is open.

2.4. Hardware Configuration — Wokwi Simulation

The Wokwi simulation files are located in `labs/wokwi/lab2.1/`. The `diagram.json` wires the Arduino Mega 2560 to a push button (Pin 7), a green LED with 220 Ω resistor (Pin 8), a red LED with 220 Ω resistor (Pin 9), and a yellow LED with 220 Ω resistor (Pin 10), exactly matching the code pin mapping. The `wokwi.toml` points to the compiled `lab2_1` firmware hex.

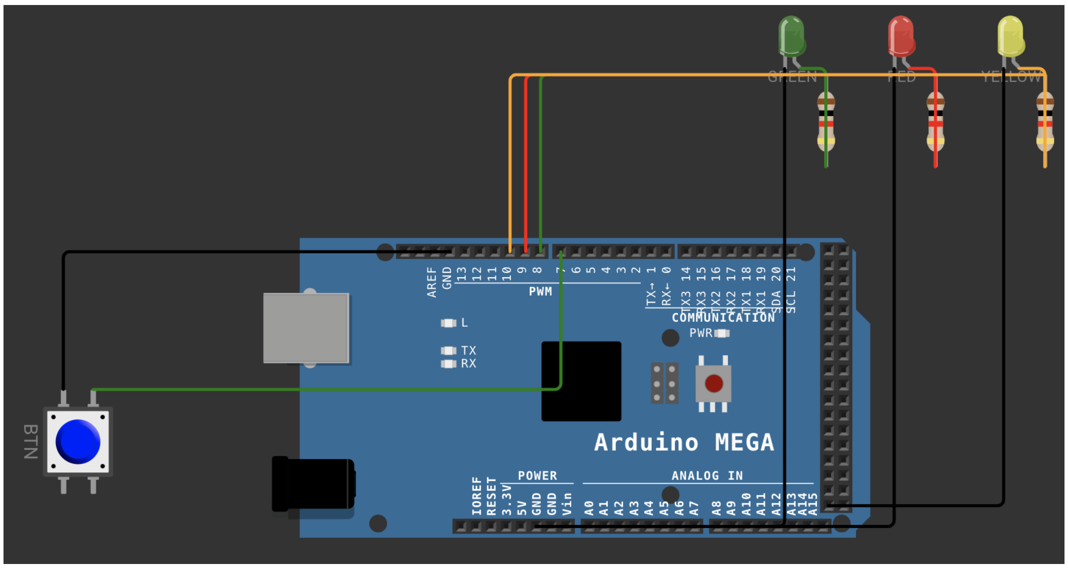


Figure 2.9 *– Wokwi simulation — Lab 2.1 circuit with button and three LEDs (active run)*

2.5. Project Structure

The project follows the established repository layout. The Lab 2.1 entry point lives in `labs/lab/lab2_1/` while the new `TaskScheduler` library is placed in `labs/lib/TaskScheduler/` to make it reusable in future labs. The `StdioSerial` library from Lab 1.1 is reused without modification. The `main.cpp` lab selector gains an additional `LAB2_1` branch, and `platformio.ini` receives a new `[env:lab2_1]` section.

Listing 2.1 *– Repository layout for Lab 2.1*

```
labs/
  src/
    main.cpp                # Lab selector (conditionally includes lab entry points)
  lab/
    lab2_1/
```

```
      lab2_1_main.h         # Lab 2.1 entry-point declarations
      lab2_1_main.cpp       # Three task bodies + lab2_1Setup / lab2_1Loop
  lib/
    TaskScheduler/
      TaskScheduler.h       # Task context struct + scheduler API
      TaskScheduler.cpp     # Scheduler implementation
    StdioSerial/            # (reused from Lab 1.1)
  wokwi/
    lab2.1/
      diagram.json          # Wokwi circuit (button + 3 LEDs)
      wokwi.toml            # Firmware path for lab2_1 environment
  platformio.ini            # Added [env:lab2_1] environment
```

2.6. Modular Implementation

TaskScheduler Module

The `TaskScheduler` library exposes two functions: `schedulerInit()` and `schedulerRun()`. Initialisation computes the absolute `nextRun` time for each task as `millis() + offset`, so tasks with different offsets start executing at staggered times even if `setup()` is called with all offsets zero. The run function implements a simple earliest-deadline-first selection over the due task subset: it iterates the entire array once searching for due tasks, picks the one with the smallest `nextRun` value (highest overdue urgency), calls its function, and advances `nextRun` by `period`. A guard prevents cascading catch-up: if the newly computed `nextRun` is already past, it is re-anchored to `now + period`.

Lab 2.1 Main Module

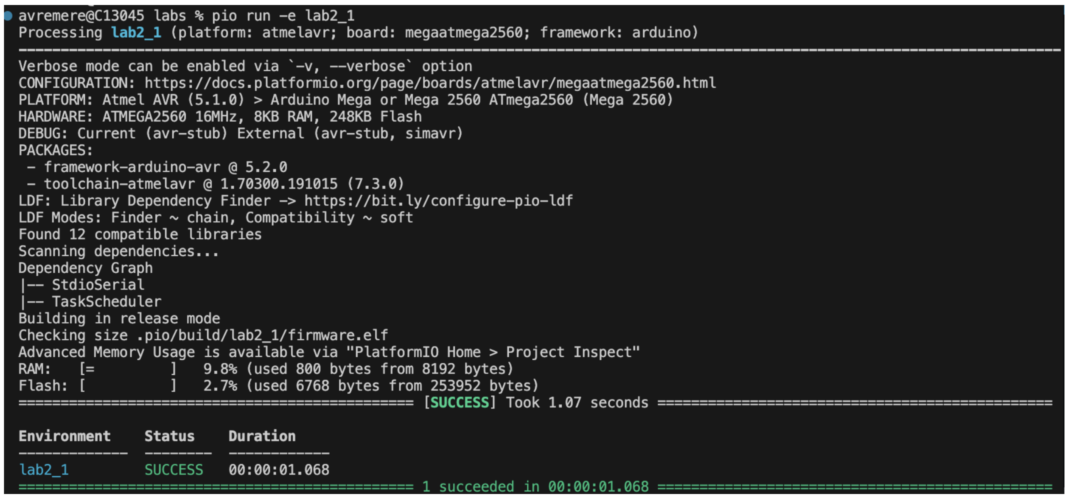
`lab2_1_main.cpp` contains all application-specific logic in three static task functions, plus the public `lab2_1Setup()` and `lab2_1Loop()` entry points. All shared state is declared as `static volatile` at file scope, making it invisible outside the translation unit. The task contexts are declared in a static array `s_tasks[]`, whose size is computed with `sizeof(s_tasks) / sizeof(s_tasks[0])` to avoid a hardcoded constant.

Task 1 uses a `ButtonState_e` enum and four `static` variables to maintain the FSM state across calls without dynamic allocation. Task 2 maintains the blink sequencer state with three variables: `s_blinkStepsRemaining`, `s_blinkLastToggle`, and `s_yellowLedOn`. Task 3 captures a snapshot of the mutable global counters into local variables before printing, ensuring the printed values are consistent even though the accumulators are reset immediately after.

3. Obtained Results

3.1. Build Process

The project was compiled using PlatformIO with the `lab2_1` environment targeting the Arduino Mega 2560 (ATmega2560). The build completed successfully with no errors or warnings, using only 2.7 % of the available Flash memory (6 768 bytes out of 253 952 bytes) and 9.8 % of SRAM (800 bytes out of 8 192 bytes).



```
avremere@C13045 labs % pio run -e lab2_1
Processing lab2_1 (platform: atmelavr; board: megaatmega2560; framework: arduino)
--------------------------------------------------------------------------------------------------------------------------
Verbose mode can be enabled via `-v, --verbose` option
CONFIGURATION: https://docs.platformio.org/page/boards/atmelavr/megaatmega2560.html
PLATFORM: Atmel AVR (5.1.0) > Arduino Mega or Mega 2560 ATmega2560 (Mega 2560)
HARDWARE: ATMEGA2560 16MHz, 8KB RAM, 248KB Flash
DEBUG: Current (avr-stub) External (avr-stub, simavr)
PACKAGES:
 - framework-arduino-avr @ 5.2.0
 - toolchain-atmelavr @ 1.70300.191015 (7.3.0)
LDF: Library Dependency Finder -> https://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 12 compatible libraries
Scanning dependencies...
Dependency Graph
|-- StdioSerial
|-- TaskScheduler
Building in release mode
Checking size .pio/build/lab2_1/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [=         ]   9.8% (used 800 bytes from 8192 bytes)
Flash: [          ]   2.7% (used 6768 bytes from 253952 bytes)
=============================================== [SUCCESS] Took 1.07 seconds ===============================================

Environment    Status    Duration
-------------  --------  ------------
lab2_1         SUCCESS   00:00:01.068
=============================================== 1 succeeded in 00:00:01.068 ===============================================
```

Figure 3.1 *– Successful PlatformIO build output for Lab 2.1 (`lab2_1` environment)*

The expected build summary is:

Listing 3.1 *– PlatformIO build summary*

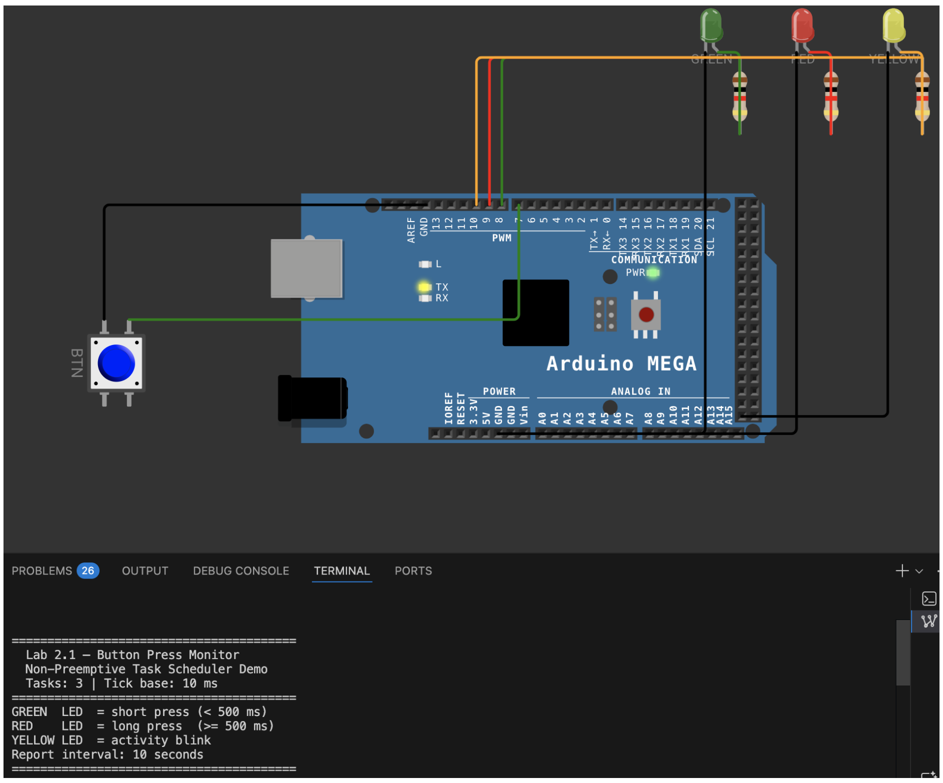
```
RAM:   [=         ]   9.8% (used 800 bytes from 8192 bytes)
Flash: [          ]   2.7% (used 6768 bytes from 253952 bytes)
========================= [SUCCESS] =========================
Environment    Status    Duration
-------------  --------  ------------
lab2_1         SUCCESS   00:00:02.xxx
```

3.2. Simulation Startup

Upon starting the Wokwi simulation with the `lab2_1` firmware, the board boots and the scheduler is initialised. Task 3 fires after an initial 2 s offset and prints the startup banner via STDIO. The serial terminal shows the welcome message:

Listing 3.2 *– Expected startup banner in the Wokwi serial terminal*

```
========================================
  Lab 2.1 -- Button Press Monitor
  Non-Preemptive Task Scheduler Demo
  Tasks: 3 | Tick base: 10 ms
========================================
GREEN  LED  = short press (< 500 ms)
RED    LED  = long press  (>= 500 ms)
YELLOW LED  = activity blink
Report interval: 10 seconds
========================================
```



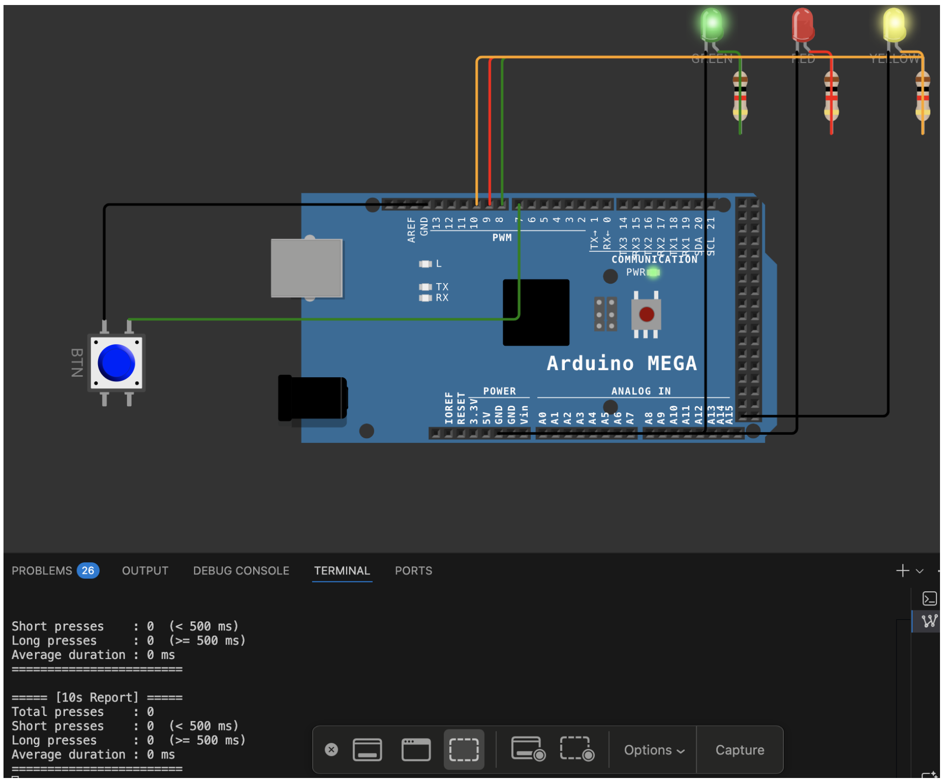
```
========================================
  Lab 2.1 – Button Press Monitor
  Non-Preemptive Task Scheduler Demo
  Tasks: 3 | Tick base: 10 ms
========================================
GREEN  LED  = short press (< 500 ms)
RED    LED  = long press  (>= 500 ms)
YELLOW LED  = activity blink
Report interval: 10 seconds
========================================
```

***Figure 3.2** – Wokwi simulation start — startup banner in serial terminal, all LEDs off*

3.3. Button Press Detection

Short Press Test (< 500 ms)

Pressing the button briefly (less than 500 ms) causes Task 1 to light the green LED for 1 500 ms and set the `g_newPress` flag. Task 2 picks up the event on its next 50 ms tick and drives a 5-blink yellow LED sequence (10 half-cycles at 100 ms each = 1 000 ms total). The green LED and yellow blink are simultaneously visible in the simulation.



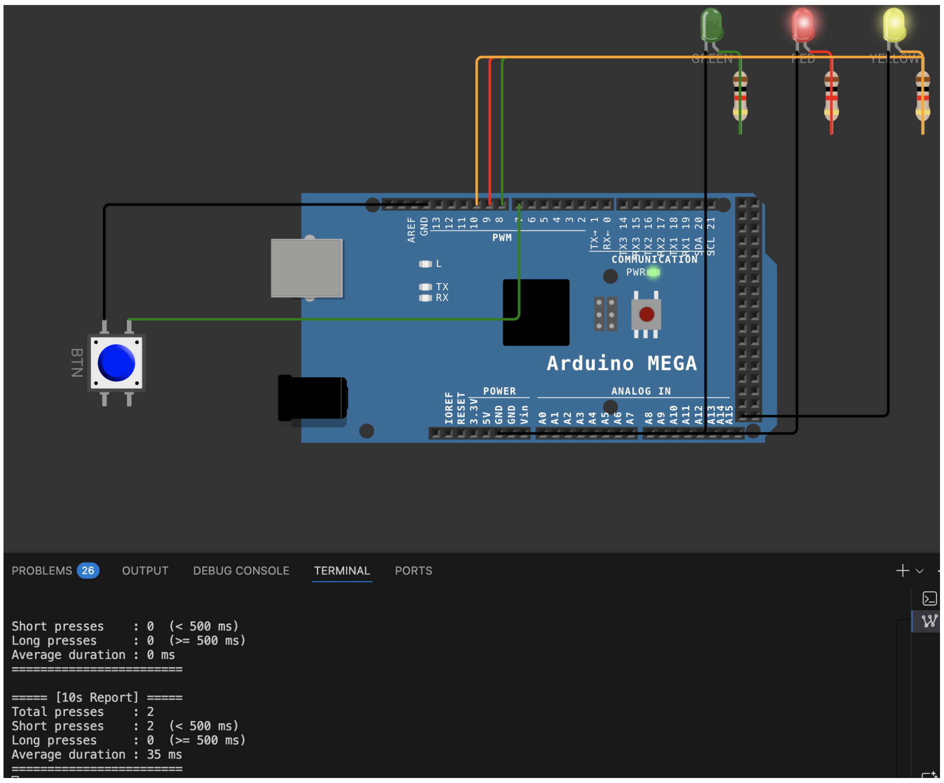
```
Short presses    : 0  (< 500 ms)
Long presses     : 0  (>= 500 ms)
Average duration : 0 ms
========================

===== [10s Report] =====
Total presses    : 0
Short presses    : 0  (< 500 ms)
Long presses     : 0  (>= 500 ms)
Average duration : 0 ms
========================
```

Figure 3.3 *– Short press result — green LED lit, yellow LED blinking (5 blinks)*

Long Press Test (≥ 500 ms)

Holding the button for more than 500 ms causes Task 1 to light the red LED and signal a long press. Task 2 starts a 10-blink yellow LED sequence (20 half-cycles = 2 000 ms). The red LED and the longer blink sequence are visible simultaneously.



```
Short presses    : 0  (< 500 ms)
Long presses     : 0  (>= 500 ms)
Average duration : 0 ms
========================

===== [10s Report] =====
Total presses    : 2
Short presses    : 2  (< 500 ms)
Long presses     : 0  (>= 500 ms)
Average duration : 35 ms
========================
```

***Figure 3.4** – Long press result — red LED lit, yellow LED blinking (10 blinks)*

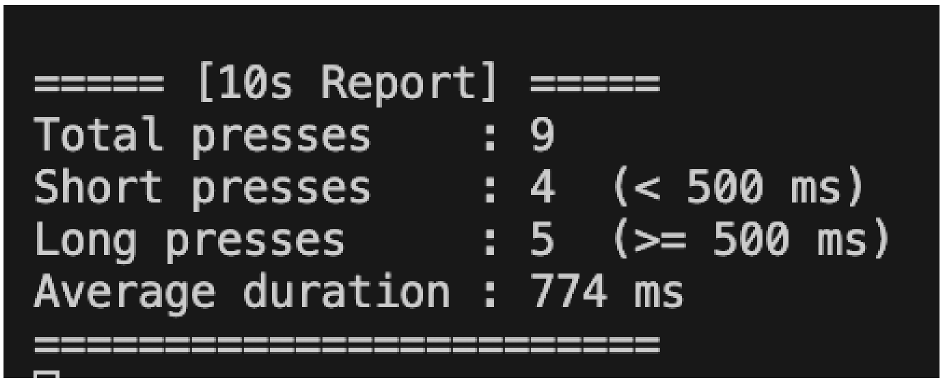
3.4. Periodic STDIO Report

Every 10 seconds, Task 3 prints the statistics for the elapsed window. After performing a mix of short and long presses, the serial terminal shows output similar to the following:

***Listing 3.3** – Example periodic report after 4 presses (2 short + 2 long)*

```
===== [10s Report] =====
Total presses    : 4
Short presses    : 2  (< 500 ms)
Long presses     : 2  (>= 500 ms)
Average duration : 712 ms
========================
```

After printing, all counters are reset to zero. A window with no button activity prints all zeroes for counts and average duration.



```
===== [10s Report] =====
Total presses    : 9
Short presses    : 4  (< 500 ms)
Long presses     : 5  (>= 500 ms)
Average duration : 774 ms
========================
```

***Figure 3.5** – Periodic STDIO report from Task 3 after a sequence of button presses*

4. Conclusion

4.1. Achievement of Objectives

All primary objectives of Laboratory Work 2.1 were successfully accomplished. A functional non-preemptive bare-metal task scheduler was designed and implemented as a reusable library, correctly dispatching exactly one task per scheduler tick based on deadline comparison. The button debounce FSM reliably distinguishes short and long presses, producing accurate duration measurements in milliseconds even under noisy mechanical contact conditions (50 ms confirmation window). Visual feedback operates correctly: the green and red LEDs illuminate for the appropriate press type, and the yellow LED produces exactly 5 or 10 blinks as specified. The periodic STDIO reporting mechanism delivers a formatted statistics summary every 10 seconds and resets accumulators to start a fresh measurement window.

4.2. Performance Analysis and System Limitations

The compiled firmware is compact, occupying approximately 6.7 KB of Flash (2.7 %) and 800 bytes of SRAM (9.8 %) on the ATmega2560, leaving ample room for future expansion. The scheduler overhead per tick is negligible: one linear sweep of a three-element array executing in a few microseconds, which is imperceptible relative to the 10 ms Task 1 period. The round-trip latency from button release to visual confirmation is bounded by one Task 1 period plus one Task 2 period — at most 60 ms — undetectable by human perception.

The main limitation of the cooperative scheduling model is shared by all bare-metal cooperative designs: if any task function blocks or executes for longer than its period, all subsequent tasks are delayed accordingly. In this application, all three tasks are short and bounded, so this is not a concern in practice. However, the `printf()` call in Task 3 imposes a small serialisation cost proportional to the output string length over a 9600 baud link (approximately 5 ms for 6 lines at 9600 baud), which is acceptable given the 10 s reporting interval. Another limitation is the integer division used for average duration: for a small number of presses with large individual durations, the integer truncation error is negligible, but it would require floating-point arithmetic for sub-millisecond precision.

4.3. Proposed Improvements

Several enhancements would strengthen the application. Adding a configurable press threshold (via STDIO command) would allow the user to tune the 500 ms boundary at runtime rather than requiring a recompile. The statistics window could be made configurable as well, since 10 s may be too short for slow manual testing or too long for high-frequency automated testing. The average duration calculation could be extended to maintain a running minimum and maximum in addition to the mean, giving the user a better characterisation of press duration distribution. For safety-critical deployments, a watchdog timer that resets the MCU if no task runs within a configurable interval would protect against task runaway. Finally, the statistics could be stored in EEPROM so that the press history survives a power cycle.

4.4. Reflections on the Learning Experience

This laboratory work provided concrete insight into how a minimal scheduling framework transforms a monolithic `loop()` into a structured set of cooperating tasks. Writing the scheduler from first principles reinforced the importance of absolute-time advancement (`nextRun += period`) over relative-time scheduling (`nextRun = millis() + period`), which accumulates drift over hundreds of ticks. The button FSM exercise highlighted how a seemingly simple user interaction — pressing a button — requires careful state management to be robust against mechanical noise. The inter-task communication pattern (one producer, one consumer, one resetter operating on shared volatile variables without locks) is a practical template applicable to a wide class of embedded sensing applications.

4.5. Impact of Technology in Real-World Applications

Cooperative bare-metal scheduling is the dominant design pattern in cost-sensitive, power-constrained, and safety-certifiable embedded products. The approach demonstrated here appears in automotive ECUs (Electronic Control Units), industrial PLCs, medical wearable devices, and consumer electronics. In automotive applications, the AUTOSAR OS specification defines cooperative and preemptive scheduling categories; many low-cost ECUs use cooperative scheduling exclusively. In medical devices, the absence of an RTOS simplifies certification under IEC 62304 because the scheduler is small, fully deterministic, and easy to review. The button-duration classification technique is directly applicable to user interaction in HVAC control panels, medical device menus, and industrial machinery where different press durations trigger different actions without requiring multiple physical buttons.

5. Note Regarding the Use of AI Tools

During the writing of this report and the development of the accompanying source code, the author used GitHub Copilot (powered by Claude Sonnet 4.6) to generate and consolidate content. The generated text, code, and diagrams were reviewed, validated, and adjusted to conform to the laboratory requirements, the teacher's specifications, and the established code architecture of the `es-labs` repository.

6. Bibliography

[1] Bragarenco, A., Astafi, V., *Sisteme Electronice Încorporate: Indicaţii metodice pentru lucrări de laborator*, Universitatea Tehnică a Moldovei, Chişinău, 2024.

[2] Atmel Corporation, *ATmega2560 Datasheet — 8-bit AVR Microcontroller with 256K Bytes In-System Programmable Flash*, Document 2549Q–AVR–02/2014, 2014.

[3] Arduino, *Arduino Mega 2560 Rev3 Documentation*, `https://docs.arduino.cc/hardware/mega-2560/`, Accessed: 2026-02-21.

[4] Kalinsky, D., *Introduction to Real-Time Scheduling for Embedded Systems*, Embedded Systems Programming, September 2001. Available: `https://www.embedded.com/introduction-to-r`
Accessed: 2026-02-21.

[5] AVR Libc Reference Manual, *Standard I/O Facilities — <stdio.h>*, `https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html`, Accessed: 2026-02-21.

[6] PlatformIO, *Professional Collaborative Platform for Embedded Development*, `https://platformio.org/`, Accessed: 2026-02-21.

[7] Wokwi, *Online Electronics Simulator — Arduino, ESP32, Raspberry Pi Pico*, `https://wokwi.com/`, Accessed: 2026-02-21.

7. Appendix

The complete source code for Laboratory Work 2.1 is organised below by architectural layer. The full project is available in the GitHub repository: `https://github.com/mcittkmims/es-labs`.

7.1. Scheduler Layer

TaskScheduler Interface

***Listing 7.1** – lib/TaskScheduler/TaskScheduler.h — Scheduler API and task context struct*

```
#ifndef TASK_SCHEDULER_H
#define TASK_SCHEDULER_H

#include <Arduino.h>

/**
```

```
 * Task context structure.
 * Holds all scheduling metadata for a single task.
 */
typedef struct {
    void     (*funcPtr)();  // Pointer to the task function. Must be non-blocking.
    uint32_t  period;       // Recurrence period in milliseconds.
    uint32_t  offset;       // Startup offset in milliseconds before first execution.
    uint32_t  nextRun;      // Absolute time (ms) of the next scheduled execution.
} TaskContext_t;

/**
 * Initialize the scheduler.
 * Computes nextRun = millis() + offset for each task.
 */
void schedulerInit(TaskContext_t *tasks, uint8_t count);

/**
 * Run one scheduler tick.
 * Executes the most-overdue due task exactly once.
 */
void schedulerRun(TaskContext_t *tasks, uint8_t count);

#endif // TASK_SCHEDULER_H
```

TaskScheduler Implementation

Listing 7.2 – *lib/TaskScheduler/TaskScheduler.cpp — Scheduler implementation*

```
#include "TaskScheduler.h"

void schedulerInit(TaskContext_t *tasks, uint8_t count) {
    uint32_t now = millis();
    for (uint8_t i = 0; i < count; i++) {
        tasks[i].nextRun = now + tasks[i].offset;
    }
}

void schedulerRun(TaskContext_t *tasks, uint8_t count) {
    uint32_t now      = millis();
    int8_t   chosen   = -1;
    uint32_t earliest = UINT32_MAX;

    // Find the most-overdue task among all tasks that are due this tick.
    for (uint8_t i = 0; i < count; i++) {
        if (now >= tasks[i].nextRun) {
            if (tasks[i].nextRun < earliest) {
                earliest = tasks[i].nextRun;
                chosen   = i;
            }
        }
    }

    if (chosen >= 0) {
        tasks[chosen].funcPtr();
        tasks[chosen].nextRun += tasks[chosen].period;

        // Guard: re-anchor if we've fallen far behind schedule.
        if (tasks[chosen].nextRun < millis()) {
            tasks[chosen].nextRun = millis() + tasks[chosen].period;
        }
    }
}
```

7.2. Application Layer

Lab 2.1 Entry Point Interface

Listing 7.3 *– lab/lab2_1/lab2_1_main.h — Lab 2.1 declarations*

```
#ifndef LAB2_1_MAIN_H
#define LAB2_1_MAIN_H

/**
 * Initialize peripherals and the task scheduler for Lab 2.1.
 * Configures button (INPUT_PULLUP), LED pins (OUTPUT), STDIO serial,
 * and registers the three tasks in the TaskScheduler.
 */
void lab2_1Setup();

/**
 * Main application loop --- drives the non-preemptive task scheduler.
 * Calls schedulerRun() every iteration; at most one task runs per call.
 */
void lab2_1Loop();

#endif // LAB2_1_MAIN_H
```

Lab 2.1 Main Implementation

Listing 7.4 *– lab/lab2_1/lab2_1_main.cpp — Three-task cooperative application*

```
#include "lab2_1_main.h"
#include <Arduino.h>
#include <stdio.h>
#include "TaskScheduler.h"
#include "StdioSerial.h"
```

```
// -- Pin mapping -----------------------------------------------------
static const uint8_t PIN_BUTTON     = 7;
static const uint8_t PIN_LED_GREEN  = 8;
static const uint8_t PIN_LED_RED    = 9;
static const uint8_t PIN_LED_YELLOW = 10;

// -- Constants -------------------------------------------------------
static const uint32_t SHORT_PRESS_THRESHOLD_MS = 500;
static const uint32_t LED_INDICATOR_DURATION_MS = 1500;
static const uint32_t BLINK_HALF_PERIOD_MS = 100;
static const uint8_t  BLINK_STEPS_SHORT    = 10;
static const uint8_t  BLINK_STEPS_LONG     = 20;
static const uint32_t DEBOUNCE_MS          = 50;

// -- Shared global state ---------------------------------------------
static volatile bool     g_newPress          = false;
static volatile uint32_t g_lastPressDuration = 0;
static volatile bool     g_isShortPress      = false;
static volatile uint32_t g_totalPresses      = 0;
static volatile uint32_t g_shortPresses      = 0;
static volatile uint32_t g_longPresses       = 0;
static volatile uint32_t g_totalDurationMs   = 0;

// -- Task 1 private state --------------------------------------------
typedef enum { BTN_IDLE, BTN_DEBOUNCE_DOWN,
               BTN_PRESSED, BTN_DEBOUNCE_UP } ButtonState_e;

static ButtonState_e s_btnState      = BTN_IDLE;
static uint32_t      s_debounceStart = 0;
static uint32_t      s_pressStart    = 0;
static uint32_t      s_pressEnd      = 0;
static uint32_t      s_greenLedOffAt = 0;
static uint32_t      s_redLedOffAt   = 0;

// -- Task 2 private state --------------------------------------------
static uint8_t  s_blinkStepsRemaining = 0;
static uint32_t s_blinkLastToggle     = 0;
static bool     s_yellowLedOn        = false;

// -- Forward declarations --------------------------------------------
static void task1ButtonAndLed();
static void task2StatisticsAndBlink();
static void task3PeriodicReport();

// -- Task context array (recurrence table) ---------------------------
static TaskContext_t s_tasks[] = {
    { task1ButtonAndLed,       10,     0 },
    { task2StatisticsAndBlink, 50,     5 },
    { task3PeriodicReport,  10000,  2000 },
};
static const uint8_t TASK_COUNT = sizeof(s_tasks) / sizeof(s_tasks[0]);

// -- Task 1: Button Detection + LED Signaling ---------------------------
static void task1ButtonAndLed() {
    uint32_t now    = millis();
    bool     btnLow = (digitalRead(PIN_BUTTON) == LOW);

    switch (s_btnState) {
        case BTN_IDLE:
            if (btnLow) { s_debounceStart = now; s_btnState = BTN_DEBOUNCE_DOWN; }
            break;
        case BTN_DEBOUNCE_DOWN:
            if (!btnLow)                                  { s_btnState = BTN_IDLE; }
            else if (now - s_debounceStart >= DEBOUNCE_MS) { s_pressStart = now;
    s_btnState = BTN_PRESSED; }
            break;
        case BTN_PRESSED:
            if (!btnLow) { s_pressEnd = now; s_debounceStart = now; s_btnState =
    BTN_DEBOUNCE_UP; }
            break;
        case BTN_DEBOUNCE_UP:
            if (btnLow)                                   { s_btnState = BTN_PRESSED; }
            else if (now - s_debounceStart >= DEBOUNCE_MS) {
                uint32_t dur = s_pressEnd - s_pressStart;
                g_lastPressDuration = dur;
                g_isShortPress      = (dur < SHORT_PRESS_THRESHOLD_MS);
                g_newPress          = true;
                if (g_isShortPress) { digitalWrite(PIN_LED_GREEN, HIGH); s_greenLedOffAt
    = now + LED_INDICATOR_DURATION_MS; }
                else                { digitalWrite(PIN_LED_RED,   HIGH); s_redLedOffAt
    = now + LED_INDICATOR_DURATION_MS; }
                s_btnState = BTN_IDLE;
            }
            break;
    }

    if (s_greenLedOffAt && now >= s_greenLedOffAt) { digitalWrite(PIN_LED_GREEN, LOW);
    s_greenLedOffAt = 0; }
    if (s_redLedOffAt   && now >= s_redLedOffAt)   { digitalWrite(PIN_LED_RED,   LOW);
    s_redLedOffAt   = 0; }
}

// -- Task 2: Statistics + Yellow LED Blink Sequencer -------------------
static void task2StatisticsAndBlink() {
    uint32_t now = millis();
    if (g_newPress) {
        g_newPress = false;
        g_totalPresses++;
        g_totalDurationMs += g_lastPressDuration;
        if (g_isShortPress) { g_shortPresses++; s_blinkStepsRemaining = BLINK_STEPS_SHORT
    ; }
        else                { g_longPresses++;  s_blinkStepsRemaining = BLINK_STEPS_LONG;
      }
        digitalWrite(PIN_LED_YELLOW, HIGH);
        s_yellowLedOn     = true;
        s_blinkLastToggle = now;
    }
    if (s_blinkStepsRemaining > 0 && (now - s_blinkLastToggle) >= BLINK_HALF_PERIOD_MS) {
        s_yellowLedOn = !s_yellowLedOn;
        digitalWrite(PIN_LED_YELLOW, s_yellowLedOn ? HIGH : LOW);
        s_blinkLastToggle = now;
        if (--s_blinkStepsRemaining == 0) { digitalWrite(PIN_LED_YELLOW, LOW);
    s_yellowLedOn = false; }
    }
}

// -- Task 3: Periodic STDIO Report -------------------------------------
static void task3PeriodicReport() {
    uint32_t total   = g_totalPresses;
    uint32_t shorts  = g_shortPresses;
    uint32_t longs   = g_longPresses;
    uint32_t totalMs = g_totalDurationMs;
    uint32_t avgMs   = (total > 0) ? (totalMs / total) : 0;

    printf("\r\n===== [10s Report] =====\r\n");
    printf("Total presses    : %lu\r\n", total);
    printf("Short presses    : %lu  (< %u ms)\r\n", shorts, (unsigned)
    SHORT_PRESS_THRESHOLD_MS);
    printf("Long presses     : %lu  (>= %u ms)\r\n", longs, (unsigned)
    SHORT_PRESS_THRESHOLD_MS);
    printf("Average duration : %lu ms\r\n", avgMs);
    printf("========================\r\n");

    g_totalPresses = g_shortPresses = g_longPresses = g_totalDurationMs = 0;
}

// -- Public entry points -----------------------------------------------
void lab2_1Setup() {
    pinMode(PIN_BUTTON,     INPUT_PULLUP);
    pinMode(PIN_LED_GREEN,  OUTPUT);
    pinMode(PIN_LED_RED,    OUTPUT);
    pinMode(PIN_LED_YELLOW, OUTPUT);
    digitalWrite(PIN_LED_GREEN,  LOW);
    digitalWrite(PIN_LED_RED,    LOW);
    digitalWrite(PIN_LED_YELLOW, LOW);

    stdioSerialInit(9600);
    printf("\r\n========================================\r\n");
    printf("  Lab 2.1 -- Button Press Monitor\r\n");
    printf("  Non-Preemptive Task Scheduler Demo\r\n");
    printf("  Tasks: 3 | Tick base: 10 ms\r\n");
    printf("========================================\r\n\r\n");

    schedulerInit(s_tasks, TASK_COUNT);
}

void lab2_1Loop() {
    schedulerRun(s_tasks, TASK_COUNT);
}
```

7.3. Application Entry Point

Lab Selector — main.cpp (relevant section)

Listing 7.5 – *src/main.cpp — Lab selector with Lab 2.1 added*

```
#include <Arduino.h>

#if defined(LAB1_1)
    #include "lab1_1_main.h"
#elif defined(LAB1_2)
    #include "lab1_2_main.h"
#elif defined(LAB2_1)
    #include "lab2_1_main.h"
#else
    #error "No lab selected! Add -DLABx_x to build_flags in platformio.ini"
#endif

void setup() {
#if defined(LAB2_1)
    lab2_1Setup();
#endif
}

void loop() {
#if defined(LAB2_1)
    lab2_1Loop();
#endif
}
```

PlatformIO Environment Configuration

***Listing 7.6** – platformio.ini — Lab 2.1 environment section*

```
[env:lab2_1]
platform = atmelavr
board = megaatmega2560
framework = arduino
monitor_speed = 9600
build_src_filter = +<*> +<../lab/lab2_1/*>
build_flags = -I lab/lab2_1 -DLAB2_1
```