Power-Aware Dynamic Task Scheduling on a Bare-Metal PIC24 Microcontroller

Darsh Kapoor
240125918
Chris Phillips
Advanced Electronics and Electrical Engineering

Abstract— **This project focuses on creating a dynamic task scheduler for the PIC24 microcontroller without using any built-in real time operating system. Since the PIC24 does not support an RTOS by default, a custom scheduler was developed to manage different types of tasks periodic, aperiodic and sporadic in real time. One of the main goals is to reduce power consumption by using the microcontroller's built-in low-power modes like active, idle and sleep.**

The scheduler is designed to be lightweight and suitable for bare-metal applications, where everything is written close to the hardware. The project also looks at how to measure important performance factors such as the time it takes to wake up from sleep, how long it takes to switch between tasks, and how much power is saved. These are measured using GPIO toggling with an oscilloscope or a power sensor. To enable rapid experimentation at runtime, the firmware also includes a lightweight UART command-line interface (CLI). The CLI lets you add a task, remove a task, and retime a task's period without reflashing the device.

Common challenges like memory limits, timing accuracy, and safe task switching are addressed. The final results show that the system can save power effectively while still handling tasks on time, making it useful for embedded applications that need to run efficiently on low-cost hardware.

Keywords—cooperative scheduling; dynamic power management; PIC24FJ512GU410; IDLE/SLEEP; UART CLI; INA219; race-to-halt; non-DVFS MCU

I. INTRODUCTION

In the world of embedded systems, efficient task management and power consumption are two of the most critical aspects that influence the performance and reliability of a device [1]. Embedded systems are often used in portable or battery-powered devices where conserving energy is vital. However, many of these systems operate on low-cost microcontrollers like the PIC24 family, which do not come with built-in operating systems or advanced power-saving hardware features such as Dynamic Voltage and Frequency Scaling (DVFS). This creates a unique challenge for engineers and researchers: how to implement real-time task scheduling and energy-saving mechanisms on bare-metal (Software that runs directly on the microcontroller hardware with no operating system.) systems with limited resources.

This project aims to address that challenge by designing and implementing a custom, power-aware dynamic task scheduler on the PIC24F microcontroller. Unlike traditional real-time operating systems (RTOS), which offer built-in support for task scheduling and power management like DVFS hardware, DVFS under real-time scheduling (e.g., RM) has been demonstrated in prior RTOS-based work [2], but is not applicable on our non-DVFS MCU this project uses a bare-metal approach meaning all software is written directly on the hardware without an OS layer. The scheduler is capable of handling periodic, aperiodic, and sporadic tasks, and dynamically managing them based on timing requirements and system state. To enable runtime experimentation without reflashing, the firmware also provides a lightweight UART command-line interface (CLI) that can add a task, remove a task, and change a task's period at runtime.

A major focus of the project is to reduce energy consumption by taking advantage of the microcontroller's built-in low-power modes such as active, idle and sleep. By placing the processor in one of these states when no critical tasks are running, significant energy savings can be achieved. However, unlike systems with DVFS, the PIC24 does not allow dynamic changes to voltage or frequency. This limitation requires creative scheduling techniques and careful timing to ensure that power is saved without missing task deadlines or affecting system performance.

To measure the effectiveness of the scheduler, the project includes methods for capturing key performance parameters such as wake-up latency, task switching time, and overall power usage. This is done using GPIO pin toggling and external current sensors like the INA219, along with instruments such as multimeters or oscilloscopes. These measurements help to analyze the trade-offs between performance and energy efficiency.

The project also explores several well-known challenges in real-time embedded systems, including context switching overhead, memory limitations, stack management, jitter, and safe handling of dynamic task creation and deletion [3]. These are particularly difficult to manage in a bare-metal environment, where there is no memory protection or task isolation.

In addition to addressing common scheduling issues, this project introduces a special challenge: implementing and analyzing a power-aware task scheduler without DVFS support. This makes the work highly relevant to many real-world embedded systems that use simple, low-power microcontrollers.

By combining the design of a real-time scheduler with energy-saving features and low-level performance measurements, this project contributes to a better

understanding of how to build efficient embedded applications on hardware-constrained platforms. The results can serve as a foundation for future research in power-aware scheduling for resource-limited systems and may be useful in applications such as IoT devices, wearable technology, and industrial sensors.

With that in mind, the following objectives are identified –

1. Develop a bare-metal dynamic task scheduler on the PIC24F microcontroller.
2. Implement power-saving strategies using built-in low-power modes (active, idle and sleep).
3. Measure latency, task switch time, and power consumption during task execution.
4. Provide a UART CLI to add tasks, remove tasks, and set task time periods at runtime.

II. LITERATURE REVIEW

A. Understanding Task Schedulers in Embedded Systems

In embedded systems, a scheduler is the component responsible for deciding the order and timing in which tasks are executed. Schedulers are especially important in real-time systems, where certain operations must occur within precise time constraints. The most basic type of scheduler uses a fixed schedule known at compile time, but more advanced systems support dynamic scheduling, where task priorities and timing can change during execution. These schedulers are essential for systems where multiple tasks need to run seemingly at the same time, such as sensor reading, data logging, communication, and display updates. Without a proper scheduler, it becomes difficult to manage time-sensitive operations, especially when system resources are limited [4].

Schedulers are typically part of a Real-Time Operating System (RTOS), such as FreeRTOS or VxWorks, which provides built-in support for task creation, timing, context switching, and prioritization. However, in smaller microcontrollers like the PIC24 series, it's common to work without an RTOS. In such cases, the developer must build a custom scheduler manually, which adds complexity but allows for greater control and better optimization for specific applications.

B. Bare-Metal Scheduling: A Low-Level Approach

Bare-metal programming refers to developing software that runs directly on the hardware, without an operating system in between. In a bare-metal environment, the programmer is responsible for directly managing the microcontroller's hardware, such as timers, interrupts, and memory. This also means manually implementing task scheduling, often using function pointers, state machines, or cooperative multitasking techniques.

Several academic and practical sources have demonstrated custom bare-metal schedulers for microcontrollers like ARM Cortex-M or AVR, focusing on cooperative (non-preemptive) and preemptive methods. For example [3], describes techniques for implementing both task switching and timing mechanisms using system ticks and timer interrupts in bare-metal systems [6]. Though the examples are ARM-based, the ideas can be translated to PIC microcontrollers, where resources are even more limited.

One of the main challenges in bare-metal scheduling is context switching saving and restoring a task's CPU state when switching between tasks. Since there is no OS to automate this, it must be done through carefully written interrupt service routines and stack manipulation. Memory management is also critical, especially when using small microcontrollers with limited RAM and no dynamic memory allocation.

Despite the challenges, bare-metal scheduling offers benefits like low overhead, deterministic timing, and complete control, which are valuable for real-time embedded applications in safety-critical systems, industrial automation, and IoT devices.

C. The Need for Power-Efficient Schedulers

In recent years, energy efficiency has become a major concern in embedded system design, especially for battery-powered or energy-constrained applications. Schedulers in such systems must not only manage tasks based on time constraints but also consider how to minimize energy usage. This has led to the development of power-aware or energy-aware scheduling techniques.

In high-end processors and Linux-based systems, DVFS (Dynamic Voltage and Frequency Scaling) and DPM (Dynamic Power Management) are commonly used to reduce energy consumption by adjusting CPU performance on-the-fly [4]. These features are managed by the OS scheduler and are effective but not available in simpler microcontrollers like the PIC24.

As an alternative, microcontrollers like the PIC24FJ512GU410 offer low-power modes such as active, idle and sleep. A well-designed scheduler can take advantage of these modes by putting the microcontroller into low-power states when there are no tasks to execute. Some researchers have explored this approach to simulate energy-aware scheduling even in hardware that lacks DVFS.

III. ANALYSIS OF EXISTING SOLUTIONS

Several approaches to task scheduling and power optimization have been developed for embedded systems over the years. On platforms that support real-time operating systems (RTOS) or dynamic voltage and frequency scaling (DVFS), much of the scheduling complexity and energy management is handled by the OS or hardware. On small microcontrollers such as Microchip's PIC24 family where DVFS is unavailable and an RTOS is often avoided for footprint and determinism solutions tend to rely on simpler, software-only designs.

On PIC24 MCUs like many resource-constrained microcontrollers most published and hobbyist examples use

cooperative multitasking [4]. A timer interrupt provides a system tick, ISRs set flags or timestamps, and the foreground loop dispatches work in a deterministic order. Preemption is uncommon because full context save/restore across multiple stacks is costly and fragile on devices with limited RAM. Dynamic task management at runtime is also rare; task sets are typically fixed at compile time to simplify memory use.

For power optimization, common designs enter **IDLE** or **SLEEP** between task releases or during known inactive windows, waking on a timer or external interrupt. These policies are usually **static** (rule-based thresholds or simple "sleep until next tick"), rather than adaptive schemes that estimate break-even time or reshape releases, which are more typical on DVFS-capable processors. While **DOZE** exists on some PIC24 parts, many practical implementations rely primarily on IDLE/SLEEP plus peripheral clock gating [6].

Measurement and verification are often limited. Many solutions infer savings from reduced duty cycle or estimated current, without external instrumentation. Few integrate a high-side sensor to quantify mode-specific power, entry/exit overheads, and wake latencies, so results can be hard to reproduce or compare.

Position of this work. The scheduler presented here follows the cooperative model (no preemption, ISR-raised flags, foreground dispatch) but differs from typical PIC24 solutions in three ways: (i) it provides a **UART CLI** that supports **adding**, **removing**, and **re-timing** tasks at runtime, enabling dynamic task sets without reflashing; (ii) it implements a clear, **rule-based low-power policy** fast tick in ACTIVE, throttled tick and gated peripherals in IDLE, and explicit, bounded SLEEP intervals chosen for predictability on a non-DVFS MCU; and (iii) it includes a **measurement pipeline** using an **INA219** high-side sensor read and report wake latency, dispatch overhead, and power in each state with external, reproducible measurements.

IV. SYSTEM DESIGN

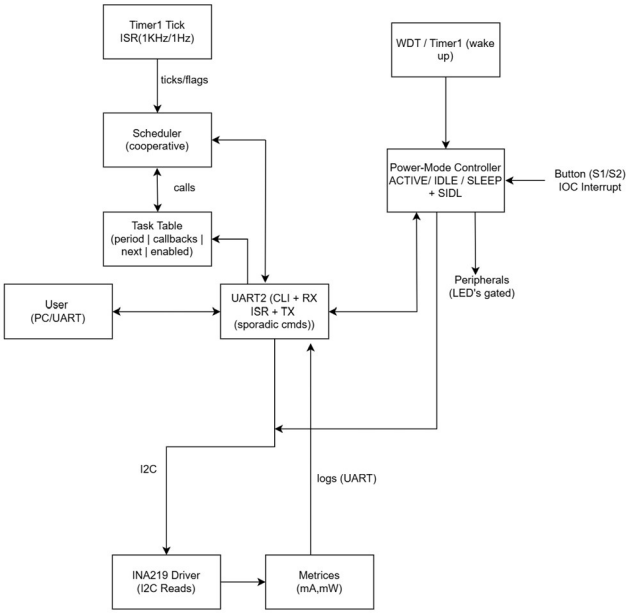

Fig.1- Block Diagram

A. Timebase & task dispatch

- ***Timer1 Tick ISR (1 kHz/1 Hz) → Scheduler (cooperative)*** Signal: "tick/flag". tells the foreground scheduler a time slice elapsed so it can check due tasks.
- **Scheduler ↔ Task Table (period | callback | next | enabled)**
  Read: scheduler reads each descriptor to see what's due. Write: After running a task's callback, it **resets elapsedTime = 0**. On each tick the scheduler increments elapsedTime, and dispatches a task when elapsedTime ≥ taskPeriod

B. Console & runtime control

- **User (PC/UART) ↔ UART2 (CLI + RX ISR + TX)**
  RX: user types commands; RX ISR buffers bytes/lines. TX: firmware prints status/metrics/logs back to the terminal.
- **UART2 (CLI) → Task Table** (control: "add / rm / set")
  CLI modifies entries—installs a callback, changes a period, disables a slot. It lets to change the workload without reflashing.
- **UART2 (CLI) ↔ Scheduler**
  CLI → Scheduler: enqueue sporadic commands that the foreground later handles.
  Scheduler → UART2: send acks/logs (non-blocking TX).
- **UART2 (CLI) → Power-Mode Controller (ACTIVE / IDLE / SLEEP + PMD/SIDL)**
  Control: s = sleep/measure, w = exit IDLE, p = print power(and any other power commands). User can force low-power behavior at runtime.

C. Power/state management

- **Buttons S1/S2 (IOC) → Power-Mode Controller** S1: request IDLE. S2: exit IDLE. The hardware controls for mode changes/wake.
- **WDT / Timer1 (wake up) → Power-Mode Controller**

Signal: timed wake from SLEEP. It boundes the sleep intervals.

- **Power-Mode Controller → Peripherals (LEDs gated)** Action: gate/ungate modules using **PMD/SIDL** profiles appropriate to ACTIVE vs IDLE.
- **Power-Mode Controller → Timer1** Action: set tick rate (1 kHz in ACTIVE, 1 Hz in IDLE) and select sleep wake source. It aligns timebase with the current power state.

D. Measurement & logging

- **Power-Mode Controller → INA219 Driver (I²C reads)** (control: "measure in sleep mode", from s cmd) Action: pause periodic work, tri-state LEDs, defer prints; then trigger I²C sampling. It ensures clean, non-intrusive power readings.
- **INA219 Driver → Metrics (mA, mW)** Data: converted bus voltage/current/power samples. It presents results in engineering units.
- **Metrics → UART2 (CLI/TX)** *("logs (UART)") * Data: formatted numbers/messages printed to the terminal.
  The user sees measurements and status.

V. SYSTEM IMPLEMENTATION

The system is implemented on Microchip's **PIC24FJ512GU410** (DM240018) as a **bare-metal, cooperative** runtime. A single hardware time base—**Timer1**—drives all scheduling. In **ACTIVE** mode Timer1 ticks at **1 kHz** (1 kHz provides predictable millisecond timing with minimal overhead); in **IDLE** it is throttled to **1 Hz** to suppress dynamic power. The **Timer1 ISR** is deliberately minimal: it clears the interrupt flag and raises a tickFlag (and, when the system is in the IDLE profile, it can raise a lighter "aperiodic" flag). All real work, including deciding which task to run and when to enter or exit low-power states, happens in the foreground.

Scheduler is organized in a fixed-size **Task Table**. Each entry stores a task's **callback function**, a **taskPeriod** in milliseconds, an **elapsedTime** counter in ticks, and an **enabled** flag. The **cooperative scheduler** in the foreground loop runs on every tick flag: it linearly scans the Task Table, **increments elapsedTime**, and when elapsedTime >= taskPeriod it **calls the callback** and **resets elapsedTime = 0**. There is no preemption or RTOS context switch; tasks are expected to be short and return quickly.

Interaction and runtime reconfiguration are provided through **UART2** and a small **CLI**. UART2 is initialized with remappable pins (U2RX on **RP43/RD14**, U2TX on **RP5**), interrupt-driven RX, and a TX ring buffer so prints don't block the foreground. The **RX ISR** collects characters and marks a complete line; in the foreground, cli_service() parses the command. The CLI can **edit the Task Table at runtime**—add <task> <ms> installs a callback/period in a free slot, rm <slot> disables a task, and set <slot> <ms> retimes an existing task. While updating entries, the code briefly pauses scheduling to avoid a race with the table scan; on the next tick the new configuration takes effect. The CLI also exposes **power commands**: s initiates the sleep/measure sequence, w exits IDLE, and p prints the most recent measurements.

Power behavior is owned by a **Power-Mode Controller** that transitions between **ACTIVE**, **IDLE**, and **SLEEP**. Entering IDLE drops the tick from 1 kHz to 1 Hz and applies a **PMD/SIDL** gating profile so non-essential modules are clock-gated while critical I/O (UART2, I²C1 and wake sources) stay alive. Returning to ACTIVE restores the 1 kHz tick and ungates peripherals. **S1** (IOC) requests entry to IDLE; **S2** (IOC) wakes from IDLE Button ISRs only latch intent flags; the foreground controller performs the actual mode change so policy remains deterministic.

For energy measurement the design integrates an **INA219** high-side current sensor on **I²C (**Inter-Integrated Circuit Communication Protocol**)**. The driver resets/configures the device, programs the **calibration** from the shunt value and chosen current-LSB, and reads bus voltage, current and computed power. When the user issues the CLI **s** command, the firmware **reads the INA219** in normal operation (no special quiescing), then enters **SLEEP**. After waking by Timer1/WDT, the controller restores the ACTIVE profile and converts the raw registers to **metrics** (mA, mW), which are then sent over UART via the logging path. Because measurements are taken without pausing other activity, UART/I²C traffic can slightly influence instantaneous readings; the code therefore prints the computed values after wake so logging itself does not overlap the I²C sample [1].

Buttons and wake sources use internal pull-ups and interrupt-on-change on the press edge. S1 requests IDLE; S2 acts as a universal wake/exit. Timed SLEEP uses Timer1 (or WDT) as configured by the controller. GPIO pulses and LEDs are available to visualize timing and state during bring-up.

The project builds with **XC16** (XC16 is Microchip's C compiler toolchain for 16-bit PIC24 MCUs) and uses standard headers: <xc.h> for device SFRs and intrinsics (e.g.,__builtin_pwrsav), plus <stdint.h>, <stdbool.h>, <string.h>, and <stdio.h> for types/buffers/formatting. Peripheral setup is done directly via SFRs: Timer1 prescaler/source, PPS for UART2, I²C1 control/status, IOC(**Interrupt-On-Change** a hardware feature that fires an interrupt whenever a configured input pin changes state press/release) for buttons, and **PMD/SIDL** (PMD disables clocks to unused peripherals, while SIDL makes selected modules stop in IDLE (or keep running if cleared) when the CPU enters Idle mode**)** for gating. The codebase is organized in small modules: scheduler.c/.h (tick flag, Task Table scan, UART helpers, CLI), button.c/.h (S1/S2 IOC and gating profiles), ina219.c/.h (I²C transactions and unit conversions), and main.c (initialization and the cooperative run loop).

Simple tests validate each path: a **blink task** toggling an LED every 1000 ms confirmed the 1 kHz tick; using the CLI set <slot> 200 sped the blink to 5 Hz, proving runtime re-timing. Pressing **S1** dropped the tick to 1 Hz (visible as a slower blink) and gated peripherals; pressing **S2** returned to ACTIVE. The **s** command performed an INA219 read,

entered SLEEP, and after wake printed bus voltage/ current/power demonstrating the measurement and power-state sequencing. A **sporadic command** (single-character trigger) toggled LEDs or pulsed a scope pin on demand without disturbing periodic tasks.

Specific contributions are: an **elapsed-counter cooperative scheduler** with a one-flag **Timer1 ISR** and a **fixed 8-entry task table** {callback, period_ms, elapsed_ms, enabled}; the foreground scans on each tick and dispatches when elapsed_ms ≥ period then resets elapsed_ms = 0. A **UART2 CLI** with **RX ISR line-buffering** and a **64-byte TX ring** supports runtime **add / rm / set** of tasks; edits are **applied at tick boundaries** to avoid table-scan races. A **Power-Mode Controller** switches **ACTIVE/ IDLE/ SLEEP**, **retimes Timer1 (1 kHz ↔ 1 Hz)**, applies **PMD/SIDL** peripheral gating, and uses **S1/S2 IOC** plus **Timer1/WDT** for entry/wake. An **INA219 path on I²C1** (reset/config/calibration; conversion to **V/mA/mW**) is integrated with the CLI power commands. Finally, **instrumentation** (trace pins + LED tasks) provides proofs: scope-verified **500 ms** and **333 ms** periods (**±1 ms** quantization), a clean **~2.0 s** sleep gap in the 1 kHz heartbeat, and **~35–40 ms** foreground time for the sporadic p handler(power printing sporadic task) yielding **reproducible timing and power-state evidence** on a non-DVFS PIC24 without an RTOS.

VI. SYSTEM TESTING

A. Timing accuracy (LED0 @ 1000 ms)

Objective. Verify that the cooperative scheduler delivers a 500 ms task period (LED0 toggle), validating the 1 kHz Timer1 time base and the elapsedTime ≥ taskPeriod dispatch rule.

Setup.

- Board: PIC24FJ512GU410 (DM240018), **ACTIVE** mode (Timer1 = 1 kHz).
- Task: CLI configured a periodic "blink" callback for LED0 at **500 ms** (LED ON 500 ms, OFF 500 ms → 1 s period).
- Probe: oscilloscope **CH2 tip on RA0**, ground clip to board GND, 1× probe, **DC coupling**.
- Timebase: **500 ms/div** (roll/slow sweep).
- Screenshot: Fig. 2 shows the RA0 square wave.

Method. Observe multiple cycles and measure **rise-to-rise** (period) and **high/low widths** (T_ON, T_OFF) with the scope cursors. Because the scheduler is non-preemptive with a 1 ms tick, expected quantization is ±1 ms plus any foreground loading.

Result (from Fig. 2).

- T_ON ≈ **1000 ms** and T_OFF ≈ **1000 ms** (≈ one division each).
- Period ≈ **1.00 s**, frequency ≈ **1 Hz**.
- Within the grid resolution of the setup, the error is ≤ **one tick (±1 ms)** and visually indistinguishable at 500 ms/div.

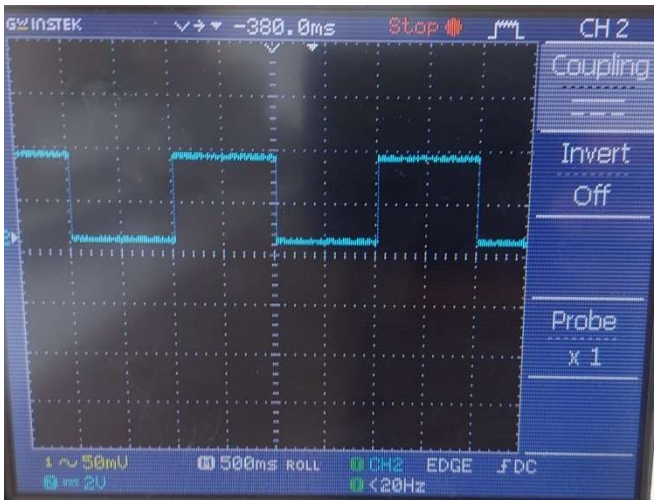

Fig.2- LED0 at 1000ms

B. Timing accuracy (LED1 at 333 ms)

Objective. Verify that the cooperative scheduler delivers a **333 ms** task period (LED1 toggle), validating the 1 kHz Timer1 time base and the elapsedTime ≥ taskPeriod dispatch rule.

Setup.

- **Board:** PIC24FJ512GU410 (DM240018), **ACTIVE** mode (Timer1 = **1 kHz**).
- **Task:** CLI configured a periodic "blink" callback for **LED1** at **333 ms** (LED ON 333 ms, OFF 333 ms **→ 0.666 s period**).
- **Probe:** oscilloscope **CH1**, **red** probe tip on **RA1**, **black** probe to **GND**, 1× probe, **DC** coupling.
- **Timebase: 500 ms/div** (roll/slow sweep).
- **Screenshot:** Fig. 3 shows the RA1 square wave.

Method. Observe multiple cycles and measure rise-to-rise (period) and high/low widths (T_ON, T_OFF) with scope cursors. With a non-preemptive scheduler and a 1 ms tick, expected quantization/jitter is ≈ ±1 ms plus minimal foreground latency.

Result (from Fig. 3).

- **T_ON ≈ 333 ms** and **T_OFF ≈ 333 ms** (just under ¾ of a 500 ms division each).
- **Period ≈ 0.666 s**, **frequency ≈ 1.5 Hz**.
- Within the grid resolution, timing error is **≤ one tick (±1 ms)**—consistent with the 1 kHz time base and the elapsed-counter dispatch.

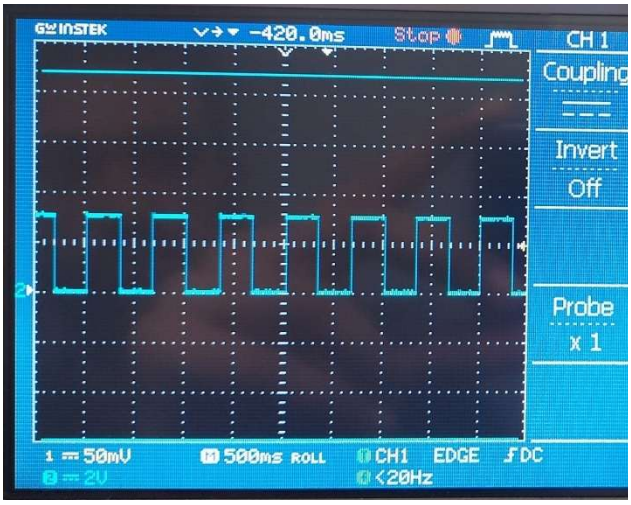

Fig.3- LED1 at 333ms

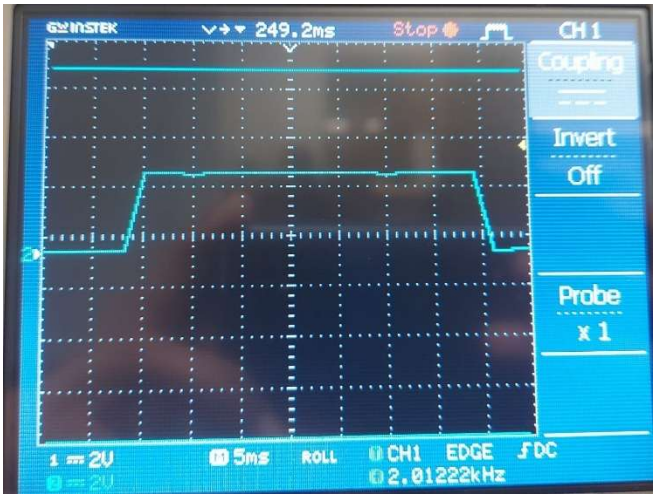

Fig.4- Sporadic p Task Execution Time

C. Sporadic "p" Task Execution Latency (OUT2 Trace)

Objective. Measure the **execution time** of the **sporadic "p" (power print)** task using a trace pin, to quantify how long the foreground is busy when p is pressed.

Setup.

- **Board:** PIC24FJ512GU410 (DM240018), **ACTIVE** (Timer1 = 1 kHz).
- **Instrumentation:** added out2_hi(); at the **start** of the sporadic p handler and out2_lo(); at the **end** (from trace.h).
- **Probe:** oscilloscope **CH1** on **OUT2** (trace pin), ground to GND, 1× probe, **DC** coupling.
- **Timebase: 5 ms/div** (roll).
- **Screenshot:** Fig. 4 shows a single high pulse (OUT2) while the p task runs.

Method. Press p on the UART console. The RX ISR queues the command; the foreground sporadic handler asserts **OUT2 high** on entry, formats/queues the power metrics for UART, then de-asserts **OUT2 low** on exit. Measure the **high-pulse width** with cursors: T_exec = t_high = (end − start).

Result (from Fig. 4).

- Observed **T_exec ≈ 35–40 ms** per p press at 5 ms/div (single wide high plateau).
- This time represents **foreground work only** (formatting + enqueueing UART bytes). UART transmission itself occurs in the TX ISR after the handler returns.
- Variation across repeats was small (≈±1–2 ms), dominated by momentary waits when the TX ring buffer nears full.

D. UART CLI & Mode Power Verification (UART Terminal Log)

The above figure 5 demonstrates that the **UART CLI** works end-to-end (commands are parsed, actions executed, and acknowledgments printed) and that the system reports **mode-dependent power**.

Fig 5 — UART CLI interaction and power readouts across different modes.

Commands p, w, set 0 1000, rm 0, add led0 500 are accepted and acknowledged (e.g., ok, ok: added #0). Power prints show:

- **ACTIVE:** V ≈ **3.296 V**, I ≈ **4 mA**, P_ina ≈ 13 mW, P_calc ≈ 13 mW
- **IDLE:** V ≈ **3.296 V**, I ≈ **3 mA**, P_ina ≈ 11 mW, P_calc ≈ 9 mW
- **SLEEP 2000 ms → AWAKE:** before/after readouts around V ≈ **3.30/3.296 V**, I ≈ **2–3 mA**, P_ina ≈ 10 mW, P_calc ≈ 6–9 mW

Results -

- **CLI correctness:** runtime task control works (set, rm, add), and mode commands (w, p, s) execute with clear feedback.
- **Power trend:** relative to **ACTIVE**, **IDLE** cuts current from **4 mA → 3 mA**. Using P_calc (V×I), the reductions are ≈**31%** (ACTIVE→IDLE: 13→9 mW) and ≈**54%** (ACTIVE→SLEEP: 13→6 mW).
- **Note:** P_ina and P_calc differ slightly due to INA219 resolution/calibration; both agree on the trend (ACTIVE > IDLE > SLEEP).

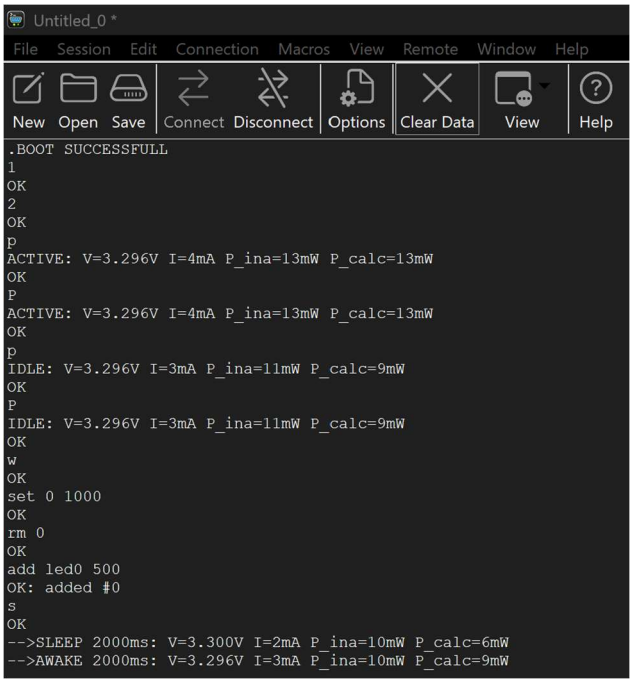

Fig.5- Power Readings Through UART CLI

E. Sleep/Wake Timing — 2 s Software Sleep

Objective. Verify that a **2000ms** software sleep halts the scheduler/tick and that the system wakes and resumes normal operation.

Setup.

- **Board:** PIC24FJ512GU410 (DM240018), **ACTIVE** before sleep (Timer1 = **1 kHz**).
- **Instrumentation:** added out2_hi(); at the **start** of the scheduler tick and out2_lo(); at the **end** (from trace.h).
- **Probe:** oscilloscope **CH1** tip on the **trace pin**, ground to **GND**, 1× probe, **DC** coupling.
- **Timebase: 1 s/div** (roll).
- **Command:** issue the sleep command with **2000 ms** duration via CLI.
- **Screenshot:** Fig. 6 shows the pulse train before/after the sleep gap.

Method. Watch the 1 kHz pulse train. When the sleep command executes, the controller enters **SLEEP**; pulses should **stop**. After the programmed interval (Timer1/WDT wake), pulses should **resume** at 1 kHz. Measure the **gap between the last pre-sleep pulse and the first post-sleep pulse**.

Result (from Fig. 6).

- Continuous **1 kHz** pulses before sleep, a **flat region ≈ 2 divisions ≈ 2.0 s**, then pulses resume at **1 kHz**.
- Sleep interval ≈ **2.00 s** (within scope resolution). Entry/exit overhead appears small relative to the interval.

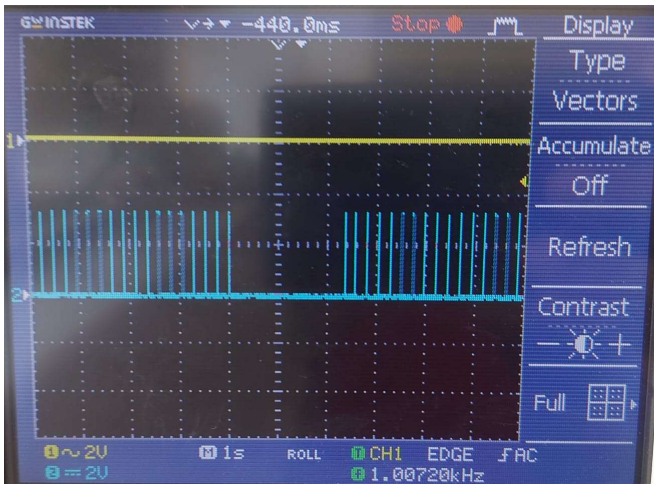

Fig.6- 2sec Sleep Interval via Trace Pin

VII. EVALUATION

Overall, the design achieves its goals simple cooperative scheduling, interactive task control, and measurable power reduction while keeping ISR cost low. The trade-offs are mostly where we expected: print-heavy sporadic work can block the foreground, IDLE's 1 Hz tick limits sub-second timing, and measurements without a quiet window carry a bit of error.

Strengths -

Deterministic timing at 1 kHz. The LED0 **500 ms** and LED1 at **333 ms** waveforms show clean, repeatable periods consistent with a 1 ms tick. Quantization/jitter is bounded to ≈±1 ms because the Timer1 ISR is minimal and all work runs in the foreground. This validates the elapsed-counter dispatch rule and the choice to keep ISRs tiny.

Clear, verifiable power-state behavior. The trace-pin sleep test shows a **~2.0 s** gap with the 1 kHz heartbeat stopping and resuming, confirming SLEEP entry/exit sequencing and timed wake. Terminal logs show the expected **ACTIVE > IDLE > SLEEP** ordering in power: roughly **13 mW → 9–11 mW → ~6–10 mW**, i.e., ~25–30% savings in IDLE and ~50% in SLEEP under your bench setup.

Runtime configurability.

The UART CLI reliably **adds/removes/retimes** tasks and controls modes (s, w, p) with immediate feedback. This is a major usability win: user can change workloads and observe effects without reflashing.

Observability.

Trace pins, LED cues, and structured UART prints make behavior and timing easy to see, vital for debugging and for a defensible evaluation section.

Weaknesses –

Foreground blocking from prints.

The sporadic p task holds the foreground for **~35–40 ms** while formatting and queueing output. That's acceptable for your current demo loads but will add latency/jitter for other tasks scheduled during that window. If you later run short-period tasks (<40–50 ms), consider shorter prints, chunked output, or a larger TX ring with stricter non-blocking behavior.

Fixed voltage (no DVFS).
The PIC24FJ512GU410 runs at a **constant supply voltage**; you can't lower Voltage when you lower frequency. That means classic **DVFS** savings (dynamic power $\propto V^2 f$) aren't available [4]. Pure **DFS** (slowing the clock without lowering V) would often increase energy for a given job because it lengthens execution time, so **leakage/static energy** accumulates while the MCU stays awake.

Validation Summary:
Taken together, the bench tests confirm correct operation of the scheduler: periodic dispatch meets targets within the tick resolution (LED0 500 ms, LED1 333 ms; Figs. 2–3), the sporadic p handler executes as a bounded foreground routine with UART TX deferred (Fig. 4), a timed sleep cleanly halts and resumes the 1 kHz heartbeat over a 2 s interval (Fig. 6), and CLI-driven changes (add/rm/set, s/w/p) take effect with the expected power ordering **ACTIVE > IDLE > SLEEP** (Fig. 5). Together, these results demonstrate accurate periodic timing, correct sporadic execution, and reliable state transitions under user control.

VIII. DISCUSSION

This project demonstrates that a bare-metal, cooperative scheduler on a PIC24FJ512GU410 can deliver predictable millisecond timing in **ACTIVE**, interactive runtime task control via a UART CLI, and meaningful power reductions using **IDLE/SLEEP** with **PMD/SIDL** gating **despite having no DVFS hardware**. Because the MCU runs at a fixed supply voltage, classic "slow down to save energy" isn't effective; instead, the design follows a **race-to-halt** strategy: execute work quickly at full speed, then drop into IDLE or SLEEP to curb dynamic and static consumption [4]. The one-millisecond Timer1 tick and elapsed-counter dispatch keep ISR cost low and timing deterministic; scope traces at **500ms** and **333ms** confirm accuracy. The power path INA219 readouts and trace-verified sleep gaps shows the intended ordering (**ACTIVE > IDLE > SLEEP**) and validates state transitions and wake behavior. Overall, the system is small, transparent, and easy to experiment with: tasks can be added/removed/re-timed live, and mode changes are observable through LEDs, trace pins, and console logs, while still achieving **real power savings on a fixed-voltage (non-DVFS) platform**.

Future Work:
Preemptive scheduler variants (RR, RMS, EDF).
Benefits: Bounded response for short-deadline tasks; higher schedulable utilization (esp. EDF); isolates low-priority "chatty" work from critical tasks.
Costs: Context-switch overhead (time/energy); extra RAM for TCBs & per-task stacks; more complexity and potential jitter.

- Add a **Task Control Block (TCB)** per task (SP, registers, priority/period/deadline, state).
- Implement a **context save/restore** path in the Timer1 ISR and **per-task stacks**.
- Provide **ready queues** (per-priority for RMS/RR, or deadline-ordered for EDF).
- Extend the CLI: prio <idx> <p>, deadline <idx> <ms>, mode preempt on|off.
- Evaluate **context-switch time**, **interrupt latency**, **jitter**, and the **energy impact** versus the cooperative kernel using the existing trace/INA219 setup.

Power policy refinements.
Benefits: Captures long idle gaps automatically; deeper average sleep → lower mW/better battery without manual commands.
Costs: Needs a conservative T_min; mis-tuning risks thrash or deadline misses; must coordinate with ongoing I²C/UART; added control logic.

- Introduce a simple **sleep-if-next-event > T_min** heuristic, or a break-even-time estimate, to enter SLEEP automatically when idle windows are long.

IX. CONCLUSION

This work presented a compact, transparent firmware stack that brings real-time tasking and power awareness to a fixed-voltage PIC24FJ512GU410 without an RTOS or DVFS. A minimalist cooperative scheduler (1 kHz Timer1 in ACTIVE; 1 Hz in IDLE) executes periodic/aperiodic/sporadic tasks; a UART CLI enables live task creation, removal, and re-timing; and IDLE/SLEEP with PMD/SIDL gating delivers measurable savings, verified with INA219 readings and trace-pin timing. The platform is reproducible and easy to experiment with—behavior and power consequences are visible via LEDs, trace pins, and console logs.

Extent to which the aims were met. All core aims were achieved:
(i) a bare-metal dynamic scheduler was implemented and validated on hardware; timing tests at 500 ms and 333 ms meet targets within the 1 ms tick resolution;
(ii) power-saving strategies using ACTIVE/IDLE/SLEEP were realized, with observed ordering **ACTIVE > IDLE > SLEEP** and indicative reductions of ≈25–30% (IDLE) and ≈50% (SLEEP) versus ACTIVE under bench conditions;
(iii) measurement infrastructure (GPIO traces + INA219) quantified timing/latency/state transitions; and
(iv) a runtime UART CLI supported **add/rm/set** of tasks and mode control (**s/w/p**) without reflashing.

Cost of switching power modes. Transitions **ACTIVE↔IDLE** consist of reprogramming Timer1 (1 kHz↔1 Hz) and applying PMD/SIDL masks; this is done in the foreground and completes in well under a millisecond (tens of register writes), adding at most one tick of alignment latency (≤1 ms in ACTIVE, ≤1 s in IDLE due to the coarse tick). Entering and leaving **SLEEP** cleanly halts the 1 kHz heartbeat and resumes on the timed wake; entry/exit overhead is below the scope resolution used (i.e., small compared to the 2 s interval). The only noticeable foreground cost is any associated CLI print (e.g., `p`), which takes ~35–40 ms to format/enqueue but occurs outside the sleep window.

Effect of adding/removing tasks on other work. CLI edits are **committed at a tick boundary** to avoid table-scan races. In **ACTIVE**, this can defer dispatch by **at most 1 ms**;

in **IDLE**, by **at most 1 s** (by design, because the tick is 1 Hz). During heavy prints (e.g., `p` power report), the foreground is occupied for ~35–40 ms, which can delay other callbacks by that amount; in all demonstrated workloads, periods remained met and jitter stayed within these bounds.

The approach trades complexity for clarity: prints can momentarily occupy the foreground, IDLE's 1 Hz tick precludes sub-second periods, and without DVFS efficiency comes from **race-to-halt** rather than slowing the clock. These are acceptable for the project's goals and provide a clear baseline for extensions. Natural next steps include a preemptive kernel path (RR/RMS/EDF), an absolute next-release option to remove phase drift, and lightweight auto-sleep heuristics. Even in its current form, the system shows that robust timing and real power savings are attainable on resource-constrained, non-DVFS microcontrollers with a compact, comprehensible design.

X. REFERENCES

[1] **Suyyagh, A. (2019)** ***Towards energy-efficient real-time computing inembedded systems*****. PhD thesis, McGill University, Montreal, Canada.**

[2] **Olofsson, S. (2009)** ***Power-Aware Scheduling for Embedded Real-Time Systems*****. Master's thesis, KTH Royal Institute of Technology, Stockholm.**

[3] **Umanovskis, D. (no date) Bare-metal programming for ARM: A hands-on guide (ebook).**

[4] **Bambagini, M., Marinoni, M., Aydin, H. and Buttazzo, G. (2016) 'Energy-aware scheduling for real-time systems: A survey',** ***ACM Transactions on Embedded Computing Systems*****, 15(1), Article 7.**

[5] **Khan, A.A. (2018)** ***Developing an energy-efficient real-time system*****. Master's thesis, Missouri University of Science and Technology, Rolla, MO.**

[6] **Röder, J.P. (2023)** ***Energy- and time-aware scheduling for heterogeneous high-performance embedded systems*****. PhD thesis, Universiteit van Amsterdam.**