ChatGPT

Benchmarking Strategy for lit-bit (Dual-Platform Support)

Safe Measurement Abstractions (Cycle-Accurate Timing)

Hardware Cycle Counters (Cortex-M & RISC-V): Use hardware cycle counters for precise, low-overhead timing on embedded targets. On ARM Cortex-M (thumbv7m, etc.), the **Data Watchpoint and Trace (DWT)** unit provides a 32-bit cycle counter (CYCCNT) that increments every core clock [1]. The `cortex_m` crate exposes this via safe APIs: for example, `cortex_m::peripheral::DWT` has methods to unlock and enable the counter and a safe `DWT::cycle_count()` function to read the current cycle count [2] [3]. To use it, one must also enable the global trace unit via `DCB::enable_trace()` (provided by the `cortex_m` crate) before enabling CYCCNT [2]. For RISC-V (e.g. riscv32imac), the **MCYCLE** CSR serves a similar purpose. The community-supported `riscv` crate provides safe wrappers like `riscv::register::mcycle::read()` (and `read64()`) to retrieve the 64-bit cycle counter without writing any `unsafe` code in your own logic [4]. These hardware counters give cycle-accurate measurements with minimal overhead, ideal for micro-benchmarks on device.

HAL Timer Abstractions: If direct use of DWT/MCYCLE is not feasible (e.g. on Cortex-M0 which lacks DWT), leverage high-level HAL constructs or timers. Many MCU HALs offer safe **monotonic timers** or stopwatch utilities built on hardware counters. For example, the STM32F4 HAL provides a `MonoTimer` / `StopWatch` that internally uses DWT CYCCNT to time intervals safely [5]. These abstractions start/stop counters or read elapsed cycles in a safe manner, encapsulating any required `unsafe` internally. Similarly, one can configure the SysTick timer or a hardware timer peripheral to measure elapsed time; the **embedded-hal** 1.0 traits (and utilities like `embedded_time` / `embedded-timers`) define a `Clock` trait for monotonic timers in no_std environments [6]. By implementing these traits for a board's timer, you get a fully safe time source (albeit with lower resolution than cycle counters) for benchmarking delays or throughput on targets without a CYCCNT.

Fully Safe Alternatives (No Hardware Counter): In cases where direct hardware cycle counters are unavailable or to avoid any reliance on target-specific registers, consider higher-level timing methods. On hosted targets (x86_64 Linux), you can use standard timers (e.g. `Instant::now()` or Criterion's timing loops) which are safe and sufficiently precise for benchmarking logic. For a deterministic measure independent of CPU frequency, you might use instruction-count based tools: e.g. the **IAI (Instruction Count) harness**. The `iai-callgrind` crate (already included in the project) runs code under Valgrind/Callgrind to count retired instructions, providing a stable metric of performance. This approach is **fully safe** for Rust code (it uses external profiling, no unsafe in your code) and works on the host to gauge algorithmic complexity or regressions. While not cycle-accurate to the hardware, it's a useful alternative for relative comparisons without embedding timers. In summary, prefer using cycle counters via well-tested crates (`cortex_m` and `riscv` crates) for on-target benchmarks [3]. Where not possible, use safe HAL timers or host-based instruction counting. All these methods avoid writing new `unsafe` and rely on either hardware or proven profiling tools for timing.

Criterion.rs Integration (Cross-Feature Benchmarking)

Host vs Embedded Benchmark Harness: To integrate **Criterion.rs** while preserving `no_std` in core logic, use a dual-crate approach. The `lit-bit` workspace already employs this pattern: keep the core library (`lit-bit-core`) lean and `no_std`, and put benchmarks in a separate crate (e.g. `lit-bit-bench`) that enables `std` [7]. The benchmarks crate can depend on `lit-bit-core` with the appropriate features (e.g. enabling `std` and async support for host tests) [8]. This way, you can run Criterion benchmarks on the host (x86_64) without pulling in heavy dependencies for end users or breaking no_std compatibility. In practice, you would conditionally compile or feature-gate any code needed only for benchmarking. For example, `lit-bit-bench` enables the core crate's `"std"` and `"async-tokio"` features in its Cargo.toml [8], allowing use of Tokio and allocation in benchmarks while the core library remains no_std by default. This setup lets Criterion and related crates (which require std) live in a non-published dev-only crate [7], preserving the core's platform-agnostic, no_std status.

Feature Matrix Bench Testing: To cover different feature combinations (e.g. sync vs async, different executors) in benchmarks, consider defining separate benchmark groups or even separate benchmark binaries for each configuration. Criterion allows grouping or filtering benchmarks by name, so you might compile the bench crate with various features and have the benchmark code select scenarios accordingly. Another approach is to use cargo features on the bench crate itself (e.g. an `"embassy"` feature in `lit-bit-bench`) to include/exclude certain tests. For instance, one could compile `lit-bit-bench` with `--features embassy` to run Embassy-related benchmarks (using `embassy-executor` for embedded context) versus with `async-tokio` for host async tests. The goal is to test the core logic under each feature flag without mixing contexts. Each scenario can use Criterion's filtering (`cargo bench --bench statechart_throughput --features async-tokio`) to run the relevant set. This matrix strategy ensures that adding, say, the `"async-embassy"` feature doesn't unknowingly degrade performance – you will have a benchmark run for it. Maintaining separate benchmark configurations also prevents the asynchronous runtime overheads from interfering across tests.

Isolating Async Runtime Overhead: When benchmarking asynchronous actors or state machines, it's crucial to separate the **framework overhead** (Tokio/Embassy task scheduling) from the actual logic under test. Several patterns can help achieve this isolation. One proven approach is to drive the actor or statechart in a minimal executor or even manually, rather than through a full multi-threaded runtime, during the benchmark. For example, if an actor's `handle` returns a small `Future` (like `core::future::Ready<()>`), you can **poll that future directly** to completion using a no-op waker instead of `tokio::block_on`. In fact, the project already did this: they replaced a heavy `block_on` call with a lightweight custom poll using a `noop_waker`, eliminating Tokio's scheduler overhead from the timing [9]. By constructing a `Context` with a no-op Waker and calling `Future::poll()` on the actor's future, the measurement only includes the actor's handling logic, not scheduler latency. Another pattern is to use synchronous single-thread executors for benchmarks (e.g. `futures::executor::LocalPool` or Embassy's minimal executor) when measuring throughput, so context switches are consistent. You can also design benchmarks to **pre-spawn** tasks or pre-allocate resources outside the timing loop. For instance, if measuring mailbox message throughput, spawn the actor and setup the channel before starting the Criterion timing loop, then inside `b.iter(|| { … })` only send messages and await responses. This ensures initialization costs or task scheduling don't skew the per-iteration timing. Additionally, use `debug_assert!` in any benchmark-only code instead of `assert!` inside tight loops [10] – this avoids panics in release benches while still catching logic errors in debug mode. Overall, carefully structure each

benchmark to measure the core operation (state transition, message send, etc.) in isolation. The **PerformanceTestKit** concept referenced in docs (with `spawn_actor_tokio` vs `spawn_actor_embassy`) can be implemented such that it sets up the actor under the chosen runtime once, and then measures just the send/receive or transition operations. This yields apples-to-apples comparisons of core logic performance across async runtimes, without one runtime's overhead dominating the results.

Maintaining `no_std` in Benchmarks: It's important that adding benchmarks doesn't force the core library to pull in `std`. The strategy above – keep benchmarks in a separate crate with `std` enabled – ensures core logic can still compile for embedded targets with no_std. Within the core crate, you might still write some benchmarking helpers (e.g. timing or allocation counters), but hide them behind cfg flags so they don't interfere with normal builds. For example, any code that uses `std::time` or logging for benchmarking can be included only in `#[cfg(test)]` or a `"benchmark"` feature. The `lit-bit` project has been careful to use feature flags to separate concerns: the `"async"` and `"async-tokio"` features bring in Tokio and related deps only when needed [11], and similarly a hypothetical `"bench"` feature could enable benchmarking hooks. By structuring code this way, **zero-cost abstractions** are preserved – the release build (no bench features) has no trace of the benchmarking logic. Criterion itself is used only in the bench harness (with `criterion_group!`/`criterion_main!` macros in the bench crate), so it doesn't affect core code at all. This means you can integrate powerful host-side tools like Criterion for analysis, but your embedded-focused library remains lightweight and free of those dependencies for end users [7].

Binary Size and Memory Analysis

Tracking Binary Size: To prevent code bloat and catch regressions in size (especially when enabling features like async), employ automated size measurements. A common approach is to use `cargo-binutils` or related tools to get the final binary size for key examples or tests. For instance, you can compile a minimal statechart example for `thumbv6m-none-eabi` (Cortex-M0) with and without the `"async"` feature and record the flash and RAM sizes (using `cargo size --bin example`). More powerfully, the community tool `cargo bloat` can analyze what's contributing to the binary. Integrating **cargo-bloat** in CI helps track size over time – e.g. using the `orf/cargo-bloat-action` on GitHub will report each PR's binary size and even break down size by crate and function, highlighting any new large dependencies [12] [13]. This is valuable to see if adding, say, an async executor causes a large jump. You can set up a baseline by running `cargo bloat` on a baseline commit (perhaps store the results as an artifact or in a branch) and then compare new runs to that. The cargo-bloat action can automate such comparisons and even fail the build if the size increase exceeds a threshold. This way, if an update inadvertently pulls in a heavy crate or the async machinery inflates code size, the CI will flag it early.

Heap Usage Monitoring: Since one of the project goals is memory safety (e.g. avoiding unexpected heap allocation in no_std contexts), it's useful to **scan for heap usage** and measure allocations. A "heap-safety scan" isn't a single built-in tool, but can be achieved via a combination of static analysis and runtime checks. Statically, you can enforce that the core crate does not depend on `alloc` unless the `"alloc"` feature is on – the project already does this by keeping heap-using features optional (e.g. using `heapless` for containers by default, and only enabling `Vec`/Box through the `"alloc"` or `"std"` features) [14]. In CI, you can ensure the no_std build passes (as done via `cargo check --no-default-features` for embedded targets [15]), which implies no accidental use of heap-allocating APIs. Dynamically, you can

incorporate a **global allocator proxy** in test or bench builds to catch allocations. `lit-bit-bench` demonstrates this with a `TrackingAllocator` installed as the `#[global_allocator]` in benchmarks [16]. This allocator wraps the system allocator and atomically counts every `alloc` and `dealloc` call [17]. Using its API, the benchmarks measure exactly how many bytes were allocated during an operation (by resetting the counts before an iteration and reading them after) [18]. For example, the **memory usage benchmarks** create many statechart instances and then query `GLOBAL_ALLOCATOR.allocated_bytes()` to ensure certain operations (like state transitions) allocate 0 bytes [19] [20]. You can integrate such checks as assertions in tests as well – e.g. a test that calls a function under a tracking allocator and asserts that net allocations remain zero. This effectively enforces heap-free operation in critical paths. Another tool to consider is the `dhat` crate (heap profiler), which can be used in test runs to report any allocations and their backtraces – useful for diagnosing if something starts allocating unexpectedly (though it requires std).

Stack Usage Analysis: Ensuring stack usage is within limits (especially for embedded) is crucial and can be done without unsafe code via static analysis tools. One recommended approach is to use `cargo-call-stack`, a whole-program static stack analyzer [21] [22]. This tool (by Jorge Aparicio) uses LLVM backend data (`-Z emit-stack-sizes`) to compute each function's stack frame size and then builds a call graph to determine the maximum stack depth of your program [22]. It works best on embedded executables (e.g. you can write a small `#[entry]` that exercises your statechart, then run `cargo call-stack --example my_app`). The output will show each function's stack usage and the total worst-case stack usage for each call path, all **without running the code**. This is particularly useful to gauge the impact of async futures on stack: since async state machines use generators, their state might reside in heap or static memory, but any recursion or deep call chains would show up. By running `cargo-call-stack` on both sync and async configurations, you can compare the stack requirements. If `cargo-call-stack` is too experimental to run on every CI, an alternative is to compile with `nightly -Z stack-sizes` and parse the `.stack_sizes` section in the ELF – but using the tool is far easier. Importantly, this approach needs no unsafe code; it's leveraging the compiler's own analysis. You might incorporate this into CI (perhaps as a separate nightly job) to prevent any unexpected large stack usage from creeping in.

Feature Impact Tracking: To catch binary size or memory overhead **from async features specifically**, adopt a strategy of building and measuring different feature combinations regularly. For example, set up CI to build the core library for a representative MCU target *with* `"async-embassy"` and *without* it, then compare the binary sizes. Over time, you can record these sizes (perhaps in a simple CSV in the repo or using GitHub Actions artifacts) to see trends. If enabling async support adds X KB of flash, that number should remain relatively stable release-to-release; a sudden jump might indicate a regression (like pulling in an unnecessary dependency). The same goes for heap usage – run a specific benchmark that exercises a typical async workload and measure allocations, comparing the count with previous runs. Since the project uses feature flags to tightly control what's included [11], you can be confident that, say, in a **non-async build,** no async executors or related code are present (thus minimal binary), whereas in an async build you accept some overhead. The key is to **measure and record** these differences so you can optimize if the overhead seems too high. Tools like cargo-bloat can even attribute how much binary size each feature brings in by analyzing functions and symbols associated with those modules.

GitHub Actions Integration (Continuous Benchmarking in CI)

CI Matrix for Dual-Platform: Setting up GitHub Actions to cover both host and embedded targets is essential. You can use a build matrix with axes for target and features. For example, one job can target `x86_64-unknown-linux-gnu` (Linux host) and run both tests and benchmarks, while other jobs cross-compile `lit-bit-core` for `thumbv6m-none-eabi` and `riscv32imac-unknown-none-elf` to ensure no_std builds succeed [15]. The embedded jobs can be limited to `cargo check` or `cargo test` (if using `cargo-embed` or QEMU for tests), due to lack of hardware in CI. The host job can run `cargo bench` on the `lit-bit-bench` crate. It's wise to include all relevant feature combinations in this matrix. For instance, the host benchmarks might be run twice – once with default (sync) features and once with `--features async-tokio` – or have separate benchmark binaries covering each (as discussed). Similarly, you might compile an Embassy-based test for a RISC-V board under QEMU if feasible. This matrix ensures coverage across the supported platforms and features in one CI workflow.

Continuous Benchmarking and Trend Storage: To get value from benchmarks in CI, use tools that store historical data and present it accessibly. One approach is using a **GitHub Action for benchmarking** that automates result collection and visualization. For example, the `benchmark-action/github-action-benchmark` can run your Criterion benchmarks, then commit the results to a `gh-pages` branch to display charts [23]. This action supports Criterion out-of-the-box and can produce an interactive graph of benchmarks over time (accessible via GitHub Pages for the repo) [24]. By integrating it, each push to main (or each PR) updates the performance graphs, so you can see trends (e.g. throughput ops/sec or latency ns) across commits. Additionally, this action (and similar ones) can **detect regressions**: it compares the latest results to the previous and if a threshold (which you configure) is exceeded – say a benchmark is 10% slower – it can make the CI fail or post an alert in the PR [23] [25]. This is a powerful gate to prevent performance degradation. Setting this up involves adding a step in the Actions workflow that runs `cargo bench` (with whatever flags needed), then uses the action to parse the output. Ensure that Criterion's output is in a format the action expects (Criterion can output JSON or CSV if configured, or the action may parse the standard output). An alternative service is **Bencher.dev**, which is a SaaS for continuous benchmarking that also integrates with GitHub – you would push benchmark results to Bencher and get trend analysis and regression alerts. Using either approach, the goal is to **automate collection and visualization** of performance data over time, so optimizations and regressions are clearly visible. This solves the issue of not just running benchmarks ad-hoc, but having a historical record.

Benchmark Artifacts and Logs: In addition to trend graphs, it can be useful to save raw data from each run. Configure CI to upload the Criterion report as an artifact (the HTML report produced by Criterion's `html_reports` feature, enabled in Cargo.toml [26]). This way, for any given run you can download the full detailed report (with statistical analysis, iteration counts, etc.) if needed for diagnosis. Likewise, you might store the JSON output of `iai-callgrind` runs to see instruction counts. For binary size and memory metrics, a straightforward approach is to use the GitHub Actions **Artifacts** or **Build cache**: e.g. after compiling the embedded examples, call `cargo size` and output the numbers, maybe even append them to a small text file artifact (or use the cargo-bloat action which comments the PR). These artifacts won't be user-facing, but they allow maintainers to pull historical data if investigating a regression. Combining these with the aforementioned benchmarking action gives a comprehensive CI: each PR will report any performance changes, and the main branch will have up-to-date performance trend graphs.

Performance Regression Gates: Enforcing performance budgets in CI should be done judiciously. You don't want minor fluctuations to fail builds, but you do want clear signals. Using the benchmarking GitHub Action's threshold feature (or criterion-compare) is a good solution – for example, configure it such that a >5% slowdown in any benchmark compared to the last main commit causes a PR check failure. This can be done via the action's config, causing a workflow failure or posting a red X with a comment listing which benchmarks regressed and by how much [25]. This turns performance into a first-class CI metric (similar to how test failures or linter failures stop a merge). Another gating strategy is less automated: you could require that any PR affecting performance must update a "baseline results" file in the repo and perhaps run a simple script to diff against the previous baseline. However, this is more cumbersome and error-prone than using existing tools. It's worth also gating on **binary size** for embedded: e.g. if the flash size of a minimal build exceeds a certain limit, or grows by more than X bytes, flag it. The cargo-bloat GitHub Action can be configured to fail if the binary size change is above a threshold. For heap usage, if you have benchmarks asserting zero allocations, then any violation will naturally fail those benchmarks (thus failing CI). In summary, integrate these performance checks just like tests – the CI should clearly indicate if a change causes slower throughput, higher latency, or a bigger binary, and in many cases prevent the change from merging until addressed.

Compliance with Project Rules (Safety and Maintainability)

Enforcing `#![deny(unsafe_code)]`: The core philosophy is that core library code should have no unchecked unsafe. This can be enforced by adding `#![deny(unsafe_code)]` at the top of each core crate (e.g. `lib.rs` in `lit-bit-core`). According to the project spec, all core crates carry this attribute and any unavoidable `unsafe` must be behind an opt-in feature flag and well-documented [27]. To ensure the benchmarking layers respect this, you should also consider adding the `deny(unsafe_code)` flag in the benchmark crate (if possible). In practice, the benchmark crate might need to use a little unsafe internally (for example, the `TrackingAllocator`'s `GlobalAlloc` impl is `unsafe` by necessity of the trait [17]). However, this usage is confined and can be audited. It's acceptable because it's part of dev infrastructure and uses `unsafe` only to call the system allocator. To keep track, you can run `cargo geiger` on the entire workspace in CI – this tool scans for usage of unsafe code in all dependencies. The goal would be to see zero unsafe in `lit-bit-core` (and ideally none in your own bench/test code either). `cargo geiger` will flag any introduced unsafe. If an unsafe is truly needed for a benchmarking trick (perhaps reading a CPU register not exposed in a crate), then isolate it in one module, mark that module with a feature (e.g. `bench_hw_cycles`) and **default it off** so it never compiles into normal builds. By reviewing the geiger report, maintainers can ensure no core code started using unsafe inadvertently. In summary, treat the `deny(unsafe_code)` as a gate in core, and for benchmarking crates use it where feasible, otherwise carefully gate any required unsafe. This policy upholds the project's strict safety guarantees even in ancillary code.

Feature-Gating Unsafe Dependencies: When introducing any dependency that is not 100% safe (internally uses unsafe or is heavy), hide it behind a feature flag or confine it to non-core crates. The project already does this with async runtime support: for example, Tokio (which contains `unsafe` internally) is an optional dependency under the `"async-tokio"` feature [11]. Users who don't enable that feature will never compile Tokio or its unsafe code. Similarly, if a benchmarking tool required an unsafe hack, it should be opt-in. A concrete strategy is to use Cargo's "dep:" feature aliasing – as shown in `Cargo.toml`, features like `"async-tokio"` bring in `dep:tokio` and other deps only when needed [28]. You could define a feature like `"bench-embedsafe"` that enables hardware-specific counters (bringing in e.g. `cortex_m` crate) for

benchmarks. This way, those dependencies (which might use some unsafe internally to access registers) are completely separated from the core and from normal usage. By **compartmentalizing via features**, you ensure that enabling a certain capability (like performance counters or heap tracking) is a deliberate choice and that all such code is excluded from production builds. Additionally, mark benchmark and test crates with `publish = false` (as already done [29]) so they are not shipped to crates.io – this prevents users from accidentally depending on dev-only code that might be less constrained about unsafe. In CI, test all combinations of features to guarantee that unsafe-heavy features don't accidentally impact other configurations. The combination of deny(unsafe_code) in core and careful feature gating in supporting crates creates a strong safety barrier: the core statechart logic remains purely safe Rust, and any "escape hatches" (like FFI calls or direct register access for timing) are opt-in, well-documented, and kept out of the default code paths [27] .

Zero-Cost Abstractions in Benchmarking: It's crucial that any instrumentation or benchmark support does not impose runtime overhead on the actual library. The design should follow zero-cost principles – if a feature is disabled, it should have no effect on performance or binary size. The steps above (separating into different crates, using `cfg` guards) ensure this: when you compile `lit-bit-core` for embedded with default features, none of the benchmark scaffolding is present at all. Even within the benchmark crate, strive to use efficient, low-level measurements that don't distort results. For example, reading a cycle counter register is just a single instruction, which is negligible and can be considered zero-cost for timing purposes. Avoid wrappers that add locking or heavy computation around measurements. The design so far shows this discipline: e.g. using atomic counters for allocation tracking incurs near-zero overhead in bench loops [30] , and using a no-op waker to poll futures avoids spinning up a scheduler thread [9] . These choices reflect a zero-cost mindset – they measure what's needed and nothing more. From a maintainability perspective, keep the benchmarking code **well-documented and isolated**. It's helpful to treat the benchmarking harness as first-class code (with its own module or utility functions) so that it's easy to modify as the library evolves. The `BenchmarkConfig` and `BenchmarkResults` utilities in `lit-bit-bench` are a good example of making a reusable, clear interface for running microbenchmarks [31] [32] . This kind of structure makes the benchmark suite easier to maintain and extend. Finally, ensure that benchmark and test code undergoes the same lint checks (perhaps allow `clippy::pedantic` on them too) so that they don't become a source of technical debt or UB – after all, dev code can also cause issues if it's unsound. By following these practices, the **benchmarking layer remains both low-overhead and clean**, upholding the project's quality standards without compromising the core library's safety or performance.

Summary of Recommendations: Use well-vetted, no_std-compatible crates for timing (`cortex_m`, `riscv`, HAL monotonic timers) to get cycle counts safely. Isolate all benchmark and async/test dependencies in separate crates or behind feature flags so core remains small and safe [7] [11] . Integrate Criterion on the host side for robust statistical analysis, but guard against measuring unintended costs by controlling the test environment (use direct polling, minimal executors, etc.) [9] . Leverage tools like cargo-bloat, cargo-call-stack, and custom allocators to track binary size, stack, and heap usage over time, plugging them into CI for automatic regression detection [22] [18] . Finally, make CI not just about passing tests, but about sustaining performance: use Actions to store benchmark results and fail the pipeline on significant regressions [23] . By following this strategy, the **lit-bit** project can confidently expand its feature set (async actors, etc.) on dual platforms while keeping performance and safety under tight control. The result will be a well-documented, continuously validated benchmark suite that guides development and assures users of the library's efficiency and reliability across both embedded and host environments.

Sources:

- Cortex-M DWT cycle counter (safe access via `cortex_m` crate) [2] [3]
- RISC-V cycle counter (MCYCLE) access via `riscv` crate [4]
- Japaric's blog on cycle-accurate timing with DWT [1]
- HAL example of safe stopwatch using DWT [5]
- `embedded_timers` crate – monotonic clock traits for no_std [6]
- lit-bit workspace dev dependencies setup (bench and tests isolation) [7] [33]
- Async feature flag separation (Tokio vs Embassy) [11]
- Fixes for benchmark overhead (no assert in tight loop; direct poll of future) [10] [9]
- TrackingAllocator for heap monitoring in benchmarks [17] [18]
- `cargo-call-stack` for static stack analysis (call graph with stack usage) [22]
- Cargo bloat action for binary size tracking in CI [12]
- GitHub Action for continuous benchmarking (storage & regression alerts) [23] [25]
- Project spec: deny unsafe in core, feature-gate if needed [27]

---

[1] Overhead analysis of the RTFM framework | Embedded in Rust
https://blog.japaric.io/rtfm-overhead/

[2] [3] DWT in cortex_m::peripheral - Rust
https://rtic.rs/1/api/cortex_m/peripheral/struct.DWT.html

[4] read64 in riscv::register::mcycle - Rust
https://docs.rs/riscv/latest/riscv32imac-unknown-none-elf/riscv/register/mcycle/fn.read64.html

[5] stm32f4xx_hal::dwt - Rust
https://docs.rs/stm32f4xx-hal/latest/stm32f4xx_hal/dwt/index.html

[6] embedded_timers - Rust
https://docs.rs/embedded-timers/latest/embedded_timers/

[7] [14] [29] [33] dependency_management.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/dependency_management.md

[8] [26] Cargo.toml
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/lit-bit-bench/Cargo.toml

[9] [10] 2025-05-30.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/PROGRESS/2025-05-30.md

[11] [28] 2025-05-28.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/PROGRESS/2025-05-28.md

[12] [13] cargo-bloat · Actions · GitHub Marketplace · GitHub
https://github.com/marketplace/actions/cargo-bloat

[15] main.rs
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/xtask/src/main.rs

[16] [19] [20] memory_usage.rs
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/lit-bit-bench/benches/memory_usage.rs

17 18 30 31 32 utils.rs
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/lit-bit-bench/src/utils.rs

21 22 GitHub - japaric/cargo-call-stack: Whole program static stack analysis
https://github.com/japaric/cargo-call-stack

23 24 25 GitHub - benchmark-action/github-action-benchmark: GitHub Action for continuous benchmarking to keep performance
https://github.com/benchmark-action/github-action-benchmark

27 Spec.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/Spec.md