ChatGPT

# Phase 05: Async Integration in the `lit-bit` Actor & Statechart Framework

## Async Traits in `no_std` Environments

Adding async support must be **opt-in** to avoid pulling in unwanted `std` or heap dependencies. The common approach is to use the `async-trait` procedural macro for async functions in traits, but by default this macro allocates (boxing the returned future) [1]. For embedded `no_std` targets, we should only enable such overhead when the `async` feature is on. Best practices include using conditional compilation to only apply `async_trait` when an allocator is available. For example, one might write:

```
#[cfg(feature = "async")]
use async_trait::async_trait;

#[cfg_attr(feature="async", async_trait)]
trait Actor {
    #[cfg_attr(feature="async", allow(boxed_future))]
    async fn handle(&mut self, event: Event);
    // ...
}
```

In the `no_std` case without the feature, the trait remains synchronous. This ensures **backward compatibility** – existing sync implementations see no change. Alternatively, we can avoid `async_trait` altogether by leveraging **associated types** (supported via GATs on stable Rust) to define an asynchronous return without allocation. For instance, define a trait with an associated `Future` type, e.g. `type HandleFuture<'a>: core::future::Future<Output = ()>; fn handle(&'a mut self, evt: Event) -> Self::HandleFuture<'a>;`. This pattern lets the compiler monomorphize and allocate the future on the stack with **zero heap usage** [2] [1]. In fact, experimental crates like `async-trait-static` demonstrate using nightly features (`type_alias_impl_trait`, GATs) to allow async trait methods **without** `Box` **or** `dyn` on `no_std` [3] [4]. Until such language features stabilize, we can gate on nightly if needed or require the async feature to be used only on recent compilers.

**Key point:** We expose async only behind a Cargo feature (e.g. `async` or `std`). This feature brings in deps like Tokio or Embassy as needed, while the core remains `no_std` and heapless. The Phase 05 design will ensure no new heap allocation or runtime overhead leaks into the sync API unless the user explicitly enables async support.

## Async Actions & Determinism in Statecharts

Introducing async behaviors into statecharts must be done carefully to **not break determinism** of state transitions. Other statechart libraries offer guidance here. **Statig**, for example, supports making state handlers and actions `async` but only when running on `std` (and with an `async` feature flag) [5]. Statig's code generation automatically detects async functions and produces an async state machine, where the `handle()` method itself becomes async (returning a `Future` that must be awaited) [6] [7]. This preserves the model of one event at a time – you must `.await` the handling of an event before sending the next. In this way, **event processing remains sequential and deterministic** even if the internal action awaits something.

One potential pitfall is so-called "async guards." In a statechart, a guard is a conditional check deciding a transition. Making a guard await an external future can introduce nondeterminism (due to uncontrolled timing and interleaving). Best practice is to **avoid awaiting inside guard conditions**. The XState community (JavaScript statecharts) calls async guards an anti-pattern and instead recommends modeling the check as a separate transitional state [8] [9]. For example, if a transition needs to wait on an async condition, the statechart should transition to an intermediate "waiting" state that invokes an async task (e.g. an API call), then on completion transitions to the appropriate next state [10] [11]. We can apply this pattern: the `statechart!` macro could provide a syntax for an *invoke state* (or using a special event) rather than a raw `async` guard. This keeps the state machine logic deterministic – the asynchronous outcome is fed back as an event (e.g. success/failure), rather than branching the transition logic mid-flight.

In summary, **async actions** (side-effectful effects performed during transitions) are supported by awaiting them as part of the state handling, but **guard conditions** should ideally remain pure or be refactored into state-driven async outcomes. This ensures the **statechart's behavior is reproducible** given the same event sequence. Libraries like **async_fsm** simply make the entire transition function async (using `async_trait`) [12] [13], which works but assumes a specific runtime (Tokio) and doesn't focus on determinism. Our approach will be to integrate async in a way that each state transition still appears atomic from the outside – any internal waits are self-contained within the transition's future. The statechart will not process a new external event while an async action is in progress, preserving logical atomicity.

## Actor Mailbox and Event Atomicity

In an actor system, ensuring one message is processed at a time (per actor) is crucial for thread-safety and determinism. When we introduce async message handlers, we must not accidentally allow reentrancy or concurrent handling of multiple messages in one actor. The actor's **mailbox loop** will therefore await the completion of each message's future before taking the next message from the queue. This pattern is analogous to how Actix's actors can handle async messages: Actix provides an `AtomicResponse` wrapper to guarantee exclusive access to actor state while a future is in flight [14] [15]. When an Actix handler returns an `AtomicResponse`, the runtime knows **not to poll the next message** until that future is done, preventing another message from interleaving and touching the actor's state [15] [16].

We will adopt a similar strategy. Internally, each actor runs as an async task that loops over incoming messages. Pseudocode sketch:

```
while let Some(msg) = mailbox.next().await {
    // Process one message at a time
    self.handle(msg).await;  // handle() may be async and will fully complete
}
```

Because we `await` the `handle` future, the actor's event loop will not advance to the next message until the current one is fully handled (including all awaits inside it). This ensures **event atomicity**: each message's side-effects happen in isolation, just as in the synchronous case. The executor is free to schedule other tasks while an actor is awaiting, but *this particular actor* won't start a new message until ready [17] [18] .

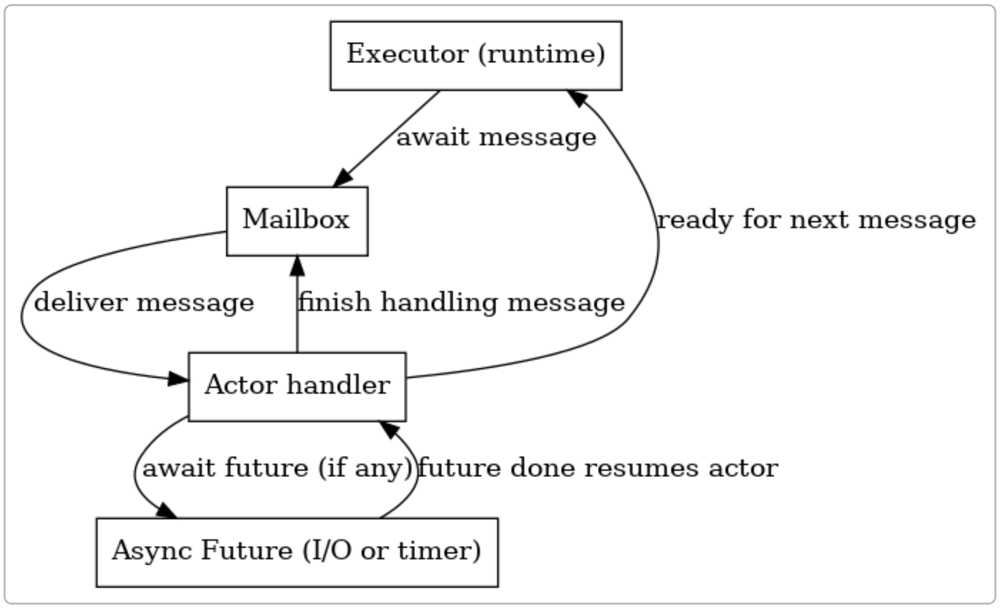

*Diagram: Actor mailbox and async message handling.* In the above diagram, the actor's mailbox delivers an event to the actor, which may call an async I/O operation (e.g. awaiting a timer or network response). The actor task yields while waiting, but the mailbox will **not dispatch another message** to that actor until the handler resumes and finishes processing. Once done, the actor notifies the mailbox (or runtime) that it's ready for the next event, preserving the one-message-at-a-time guarantee.

To implement this safely, our mailbox can use a futures-aware channel (e.g. Tokio's `mpsc` or Embassy's channel) that the actor loop awaits on. We might also introduce an explicit `AtomicMode` for critical sections if needed (similar to Actix's concept) – but if we simply always treat the entire handler as atomic, that suffices. The runtime (Tokio or Embassy) will drive the actor future, and we ensure that future doesn't complete until the message and all its sub-futures complete. This design yields **sequential message processing** with no deadlocks (the runtime isn't blocked, only the actor task is waiting) [19] .

## Backwards Compatibility via Conditional Compilation

To maintain backward compatibility, we will heavily use **conditional compilation (`cfg` flags)** and careful trait design. The existing sync API should remain unchanged when the async feature is off. Strategies to achieve this include:

- **Feature-gated extension traits:** We can define separate traits or impl blocks for async functionality. For example, a base trait `Actor` with a sync `handle()`, and if `feature = "async"` we provide an `impl ActorExt for Actor { async fn handle_async(...) { ... } }`. However, this could complicate the API. A cleaner approach is to use a single trait and make it *polymorphic* – for instance, using the associated future pattern described earlier. When `async` feature is off, we can implement that trait such that `HandleFuture<'a>` is a simple immediate future (or uses `core::future::ready()`), effectively making `handle()` sync. When `async` is on, the same trait is implemented for cases where the user's `handle()` is actually async.

- **Macro-based duplication:** Another approach is using a macro to generate both sync and async versions of functions depending on a feature flag. The [`maybe_async`] crate exemplifies this: you annotate functions or impl blocks with `#[maybe_async::maybe_async]`, and it expands to an async version or a sync version based on whether an `is_async` feature is set [20] [21]. RSpotify (a Rust Spotify client) successfully used this to offer both blocking and async APIs in one codebase [21] [22]. We might not directly use `maybe_async` (to minimize dependencies), but the concept can be integrated: e.g., the `statechart!` proc-macro can generate different code if an async feature is enabled.

- **No change to default behavior:** Crucially, if the user doesn't enable any async features, the crate will compile in pure `no_std` mode exactly as before. All new async code paths will be under `#[cfg(feature="async")]` (or subdivided into `async-tokio`, `async-embassy` features as appropriate). This ensures *no breaking changes*. Even things like timers or guards that get added will have sync fallbacks or simply be unavailable without the feature (with clear compile errors if used).

For example, we might introduce an `Actor::start()` that runs the actor on an async runtime. That function would only exist when `cfg(feature="async")` and would be hidden otherwise, so existing embedded users (who call a sync `run()` or use it in interrupt context) are unaffected. The conditional compilation will also be applied to dependencies: e.g. `tokio` and `async-trait` will be in `Cargo.toml` under optional features, so they won't be pulled in for `no_std`. This follows the pattern of many cross-platform crates (see **picoserve** HTTP server, which has separate features for `std`, `tokio`, and `embassy` integrations [23] [24]). By using **feature flags and optional deps**, we ensure a platform-dual design where the crate can be built in a minimal form or a richer async form. Users explicitly opt in to async support on their platform of choice.

In summary, careful use of `#[cfg]` keeps the crate dual-compatible. We will document that enabling `async` may pull in additional requirements (like an allocator or OS integration), but those are strictly opt-in. This way, **backwards compatibility is guaranteed**: all existing synchronous code will compile and run as before, and even binary size/runtime overhead remains the same unless async is activated.

## Multi-Runtime Support: Tokio and Embassy Integration

Phase 05 must support both a standard OS async runtime (likely **Tokio** on x86_64) and an embedded runtime (likely **Embassy** on Cortex-M/RISC-V). These two have very different models (Tokio is multithreaded, uses heap; Embassy is single-threaded `no_std`, statically allocated). We will adopt an **execution abstraction layer** to accommodate both. Several strategies are possible:

- **Compile-time selection via features:** We can provide separate modules or integration points for each runtime. For example, an `actor_runner::tokio` module that contains `impl ActorRunner for tokio::Handle` or a function `start_actor_tokio(actor, mailbox)` that uses `tokio::spawn`, and similarly an `actor_runner::embassy` that uses `embassy_executor::Spawner`. The user (or the library internally, based on target) picks which to use. The crate **picoserve** uses this approach: it defines a `tokio` feature and an `embassy` feature, each enabling the appropriate dependencies (tokio or embassy-net) [24] [25]. In our case, we might have a single `async` feature that enables both, but more fine-grained features (`async-tokio`, `async-embassy`) could be used if one wants to include only what's needed.

- **Unified `Spawn` trait:** Alternatively, we could abstract over the concept of an executor by defining a small trait (similar to `futures::task::Spawn`) that has a method to spawn a `!Send` future (for embedded) or a `Send` future (for threadpool). However, since Tokio requires `Send` futures for `tokio::spawn` (unless using `LocalSet`), whereas Embassy does not require `Send` (it runs on a single core), a single trait would need to be generic over the future's Send-ness or we provide two functions (spawn and spawn_local). Simpler is to not unify too hard, but rather provide conditionally compiled implementations.

Concretely, the actor system might provide an API like:

```
#[cfg(feature="tokio")]
pub async fn start_actor<A: Actor>(actor: A) {
    tokio::spawn(async move {
        actor_main_loop(actor, tokio::sync::mpsc::channel());
    });
}

#[cfg(feature="embassy")]
pub fn start_actor_embassy<A: Actor>(spawner: embassy_executor::Spawner, actor:
A) {
    spawner.spawn(actor_main_loop(actor,
embassy::channel::Channel::new())).unwrap();
}
```

Here `actor_main_loop` is our internal async function that awaits messages and calls the actor's handler. The above pseudo-code shows that with `tokio` we spawn a task (which requires `A: Send` perhaps), and with `embassy` we spawn via the provided `Spawner`. Embassy tasks must be `'static` and are often defined via the `#[embassy_executor::task]` macro. We can hide these details by providing the user an attribute or macro to declare their actor tasks appropriately.

The **execution model differences** also include time and blocking: Tokio uses a reactor and timer that rely on OS, whereas Embassy uses hardware timers and interrupts. We therefore ensure that any blocking or waiting in our code is done via futures (so that on Embassy it yields to the executor loop, rather than truly blocking). For example, using `Timer::after` (Embassy) or `tokio::time::sleep` for delays (see Timer section below).

A challenge is that **Tokio tasks typically require** `Send` futures because they might run on a thread pool. If our actor or its messages are !Send (e.g. contains `!Send` hardware references), we cannot use `tokio::spawn` directly. In such cases, we could require using `#[tokio::main(flavor="current_thread")]` or `tokio::task::LocalSet` to allow !Send futures, or simply document that if you use Tokio multi-threaded, your actor must be `Send`. Embassy has no such restriction. We likely target Tokio's current-thread scheduler for simplicity (similar to how one might run on a single thread for better determinism). This will be clarified in docs.

In summary, Phase 05 will support two execution models: - **Tokio (std)** – Feature enables Tokio, uses its executor for actors. Suitable for cloud or desktop usage. - **Embassy (no_std)** – Feature enables Embassy, uses its executor or requires user to start tasks. Suitable for microcontrollers.

We will provide conditional implementations and example bootstraps for both, ensuring that internal code uses portable futures and channels. The design mirrors what real projects do to support multiple runtimes: for instance, the **BackON** crate selects a sleep implementation via features (`tokio-sleep`, `embassy-sleep`, etc.) [26] [27], and networking crates may offer both `tokio` and `embassy` adapters. Our framework will be dual-target by construction, fulfilling the platform-dual architecture requirement.

## Timer Abstractions for Embedded and Cloud

Statecharts often need time-based transitions (e.g. *after 5 seconds in state X, go to Y*). We will introduce a **timer abstraction** that works in both environments. A simple approach is to define a trait or module that provides an `after(duration)` future. For example:

```
// Pseudocode
pub trait TimerService {
    type DelayFuture: core::future::Future<Output=()>;
    fn after(dur: Duration) -> Self::DelayFuture;
}
```

Then we provide implementations or functions using it: - On `std` (Tokio enabled): `TimerService::after(dur)` uses `tokio::time::sleep(dur)` (which returns a future that completes after the duration). Tokio's time feature will be enabled via `tokio::time` [26]. - On `no_std` (Embassy): use `embassy_time::Timer::after(dur)` which is an async delay that leverages a hardware timer (Embassy's time driver) [25]. Embassy's timer is cooperative and integrated with its executor's timer queue [28].

Users of the statechart might then write in the DSL: `after(1.sec()) / [action]`, which under the hood schedules a timer. We will likely implement this by scheduling a **deferred event**. For example, when entering a state with a timed transition, the framework will call `TimerService::after(d)` and then send a special event (like `Event::Timeout(StateX)`) when it fires. This event is handled by the statechart to perform the transition. This design is similar to how statechart modeling tools implement timeouts (often as an implicit event). It keeps the mechanism portable: on embedded, `Timer::after` uses no heap and leverages hardware timers; on cloud, it uses the Tokio reactor.

There are existing abstractions we can draw from: - **Embassy's** built-in timers: Embassy's executor description notes that it has an *integrated timer queue* and you can simply do `Timer::after_secs(1).await` anywhere in an async task [28]. We can rely on that for embedded. - **Futures Timer**: In a `no_std` context without Embassy, one could use the `futures-timer` crate, which provides a `Delay` future (though it spawns a thread on std – not applicable in bare metal). However, since we plan to use Embassy for embedded, we likely won't need an additional crate. - **RTIC (Real-Time Interrupt-driven Concurrency)**: In purely bare-metal scenarios, one might use an RTOS or RTIC scheduling for timeouts, but integrating that is out of scope for now; Embassy covers our needs.

By abstracting timers, we allow the `statechart!` macro to offer a high-level syntax. For example, the macro might accept something like:

```
state Foo {
    on Entry do { start_timer(1000_ms); }  // pseudo-code to set a timer
    on Timeout => transition to Bar;
}
```

Under the hood, `start_timer(ms)` would use `TimerService` to schedule an event. The user could also directly await a timer in an action: e.g. `action = async { Timer::after(ms).await; do_something(); }` if appropriate.

The key is that our timer solution must work **without** `std`. Embassy's implementation is ideal: it requires no heap and no OS, just a hardware timer (or a systick). It's proven that if tasks don't fit in RAM or too many timers, it fails at compile/link time rather than unpredictably at runtime [29]. On the cloud side, Tokio's timers are robust and efficient. Thus, the framework's timer API will be a thin shim over these platform-specific underpinnings, chosen via `#[cfg]`.

For completeness, we note that if a user runs the async version on a desktop without Tokio (say, using async-std or smol), they could in theory plug in a different `TimerService` by implementing our trait. But officially, we'll support Tokio for std and Embassy for no_std.

## Managing Future Memory Without Dynamic Allocation

Rust's async/await is zero-cost in the sense that **the compiler generates state-machine structs on the stack** for `async fn` (no heap unless you box). However, when you have long-living tasks (like actors) or many tasks, you must allocate their futures somewhere. In a no_std environment without a heap, the strategy is to use **static allocation** for futures. We will leverage patterns from Embassy, which is specifically designed for this:

Embassy tasks are **statically allocated in memory** – each task gets its own static stack frame with the exact size needed for that future, computed at compile time [29]. If you spawn too many tasks or one that is too large for RAM, it's a compile/link error, not a runtime crash [30]. Embassy achieves this with its procedural macro that wraps the async task in a `static`. We can emulate this by requiring that all actor tasks be defined either via our macro (which can create a static) or by the user providing a `StaticCell` for each actor.

For example, we might ask users on embedded to declare their actor with a macro that expands to:

```
static ACTOR_TASK: StaticCell<embassy_executor::Task<()>> = StaticCell::new();
embassy_executor::Spawner::spawn(ACTOR_TASK.init(move ||
actor_main_loop(my_actor)));
```

This is roughly how Embassy spawns tasks without dynamic allocation (it uses an internal arena if you use the dynamic `Spawner.spawn()` API, which is a simple bump allocator of a fixed size) [31]. We will aim for a design where **each actor's stack usage is fixed and known**. The `statechart!` macro can help here by perhaps declaring the state machine handling future as `#[embassy_executor::task]` when appropriate.

Additionally, all communication primitives will use **fixed-capacity buffers** from the `heapless` crate. For instance, the actor mailbox might be a `heapless::Vec` or `heapless::spsc::Queue` of events. `heapless` ensures no heap usage and predictable memory footprint. We already rely on `heapless` in core; we'll continue to do so for any async channels in `no_std`. If multiple producers are needed (e.g. multiple actors sending to one mailbox), `heapless::mpmc::Q` (multi-producer) can be used with a static allocation.

Safety is paramount: we continue to forbid `unsafe` in core logic, so we will lean on these well-tested abstractions rather than rolling our own pointer juggling. The memory model should be such that **all futures have a bounded lifetime and known storage**: - Actor tasks: static or stack allocation (none allocated on heap at runtime). - Timer futures: on Embassy, the timer queue is static; on Tokio, the timer is heap (inside Tokio), but that's hidden behind Tokio's allocator. If the user runs on an OS, presumably heap is okay. - Message futures (if any): e.g. if an actor sends an RPC and awaits a response, that response future must be pinned. The user can use `heapless` data or if they use `async_trait`, one allocation per call (which we warn about).

It's worth noting that as of 2025, we expect further improvements like **async functions in traits** to become stable (there was an MVP in nightly by late 2022 [32]). Once stabilized, we can drop the `async_trait` proc-macro and rely on the compiler's native support, which will allow trait async methods with **zero-cost** (no boxing) [1]. That will further ensure no hidden heap usage.

In summary, Phase 05 will manage futures such that on embedded targets, **no dynamic memory allocation occurs** at runtime. Memory for tasks is determined at compile time (using static buffers or compile-time sizing). This yields deterministic memory usage. Even on the cloud side, the overhead of boxing (if any) can be measured, and users concerned with absolute performance can use the newer patterns or opt to stick to sync in tight spots. As one forum noted, you *can* have Rust async with no heap allocation – the compiler does it for free as long as you don't use trait objects for futures [33] [34]. We design our traits and generics to exploit that property.

## Performance Benchmarking Plan (CI-Friendly)

With the introduction of async and new code paths, we need to ensure performance remains acceptable and catch regressions. We will implement a two-pronged benchmarking strategy: one for the cloud (hosted, full `std` environment) and one for embedded.

**Host (Cloud) Benchmarks:** We can use **Criterion.rs** for microbenchmarking certain operations (e.g. message throughput, state transition latency). Criterion provides statistical rigor and works on stable Rust. For example, we could benchmark how many simple messages an actor can process per second in sync vs async modes. Additionally, tools like `cargo bench` with Criterion can run in CI (though care is needed with noise – we might allow high variance). We will also use `cargo-bloat` and `cargo-size` to monitor code size and memory usage changes [26]. Cargo-bloat can report the top contributors to binary size; we will run it with and without async feature to quantify the overhead of including Tokio/Embassy. This can be part of CI, failing if the size exceeds a certain threshold. We should also include **heap usage** in benchmarks: e.g. measure allocations using `tracker` or enable `tokio`'s metrics for tasks if possible.

**Embedded Benchmarks:** For embedded targets (Cortex-M, RISC-V), we can't use Criterion easily on target, but we can measure cycle counts and timing using hardware counters: - On Cortex-M: use the **DWT (Data Watchpoint and Trace) cycle counter**. Many MCUs have a CYCCNT register that increments every clock cycle [35]. We can set this counter to 0, run a piece of code (e.g. process N messages), then read the counter. This gives cycle-accurate performance measurements. The `cortex_m` crate provides an API to enable and read DWT [36]. - On RISC-V: read the `mcycle` **CSR** (or `cycle` in user mode if available) which counts clock cycles [37]. The `riscv` crate exposes this register for RISC-V targets. We might write small inline assembly if needed (`csrr` instruction to read cycle count). - We will integrate these into our test or example firmware. For instance, an integration test could toggle a GPIO at start and end of a benchmark section, and use a logic analyzer – but that's not easily automated. Instead, directly reading the cycle counter in code and perhaps outputting the result via semihosting or defmt log can allow CI (with an emulator or hardware-in-the-loop) to capture it.

We can automate such tests using QEMU or real hardware in CI: - **QEMU**: QEMU can emulate ARM and RISC-V MCUs and may allow reading the cycle counter (though not 100% cycle accurate, it's a rough proxy). Still, QEMU can run the firmware and we can log cycle counts for operations. - **Hardware CI**: Using a tool like Probe-run with defmt, we can execute on a connected board. Defmt test harness can output test results (pass/fail) with embedded logging. We could embed assertions like "processing 1000 messages took < X cycles" to ensure we meet performance targets.

Additionally, we'll measure **stack usage** on embedded, since async can increase stack demand (state machines for futures). We will use **stack painting** technique for this: fill the stack memory with a known pattern and then after running, check how much of that pattern remains untouched [38] [39]. Memfault's blog describes this method – essentially, before starting the tasks, we write 0xAA...AA across the stack region, then after execution we scan for the boundary between overwritten and untouched memory [40]. This can be done in a test and reported. We can automate it by linking a symbol at stack start/end and doing a small routine in the firmware.

Finally, we'll incorporate **profiling of code size and cycles for critical sections** as part of CI. For RISC-V specifically, there's mention of a `riscv_perf` – likely referring to hardware performance counters or perhaps a crate. We will simply use the RISC-V performance counters for cycles and maybe instructions retired.

The benchmarking plan is thus: - **Criterion on x86_64**: for high-level throughput tests (function call overhead, etc.). - **cargo-bloat**: monitor binary size differences (especially important that enabling async doesn't blow up code for embedded beyond acceptable limits). - **Cycle count tests on embedded**: using DWT (ARM) and MCYCLE (RISC-V) to ensure the overhead per message or per transition is within our budget. - **Stack usage analysis**: using stack painting or static analysis if possible, to detect any runaway stack growth from async futures.

All these should be **CI-friendly** (non-interactive, automatable). We will likely use a nightly toolchain or `cargo callstack` for static stack analysis in CI to complement runtime checks. The goal is to catch performance regressions early – e.g., if a change causes an extra heap allocation or doubles the cycle count of a transition, CI should flag it.

## Phase 04 Deferred Features Unblocked by Async

Several features postponed in Phase 04 can now be implemented or improved thanks to the async infrastructure:

- **Supervisor and Actor Lifecycle Management:** Phase 04 likely envisioned a supervision tree (actors monitoring/restarting others) but implementing this robustly might have been deferred. With async, we can model actor death and restart as asynchronous events. For example, a supervisor can `await` the termination of a child actor's future. In an async context, if a child actor task ends (returns or panics), the supervisor task can get notified (e.g. via a `JoinHandle` or a channel). This enables OTP-style supervision where a supervisor automatically restarts a child. The **Ractor** framework demonstrates this: it links actors in a parent-child relationship and the supervisor is notified on child stop/failure [41]. They even have tests for supervision behavior [42]. We will implement similar capabilities: an actor can spawn a child actor (as a new task) and `.await` a handle for it or receive a message on failure. This was hard to do in pure sync (would have required polling threads or callback hooks), but async makes it straightforward to **await child completion**. We will also add test support for this: e.g., a test can spawn a supervisor with a failing child and assert that the supervisor took the correct action (restart, log, etc.).

- **Message Batching and Backpressure:** With async message handling, we can introduce optimizations like batching. For instance, an actor could choose to pull multiple messages from the mailbox in one go if the system is busy, to amortize wakeup overhead. In sync code, you might loop until queue empty; we can do the same in async by reading from an `mpsc` in a loop without awaiting in between until certain conditions. Also, we can leverage async to implement **backpressure** or **throttling**: if an actor is slow to process, the mailbox `send()` futures can await when the queue is full (in embedded, `heapless::MPMC` can indicate full). Batching was deferred perhaps due to complexity, but now we can add an API like `handle_batch(&mut self, messages: &mut [M])` or simply document that the actor should loop to handle all pending messages when woken. We will explore using Tokio's ability to tune poll behavior or Embassy's channel `try_recv` in a loop to empty the queue on each wake. This improves throughput and was probably slated for Phase 04.

- **Test Probes and Deterministic Testing:** Async opens up new ways to **observe actor internals for testing**. We can create test harness futures that wait for certain conditions in the actor. For example, a test can send an event to an actor and then `.await` a notification (perhaps the actor emits a special probe event or writes to a watch channel). In sync, one might have had to spin on a condition or use hooks, but in async we can `await` on multiple events. We will provide a **test support module** that might include an instrumented mailbox or a way to register a probe callback (e.g., an async channel that the actor can send state snapshots to). The end-to-end deterministic execution is easier when the whole system is single-threaded async: we can simulate an event loop in tests and use `tokio::test` or `embassy_executor::run` in a controlled environment. Features like *trace logging of state transitions* (deferred before) can now be done by simply awaiting on an internal channel of trace events.

- **Message Priorities and Cancellation:** Perhaps Phase 04 mentioned message priority or cancellation (like aborting a pending transition). With async, implementing cancellation is natural via `Future::cancel` (dropping the future) or using `select!` to race events. For instance, we could implement a **timeout** for a message handling using `tokio::select!` (or Embassy's `select`) between the message future and a timeout future. This could realize patterns like *if an actor does not complete a task in X time, consider it failed* – relevant to supervision or simply as a feature.

All these enhancements are enabled or simplified by having the async runtime in place. The plan is to review Phase 04's deferred list and tackle those which align with async capabilities first. Based on the prompt, **supervision, batching, and probes** are top candidates: - We will implement a basic **Supervisor** actor that can restart child actors (with an option to limit retries or escalate). - Implement **batch processing** option in mailboxes (perhaps when an actor signals it's willing to handle a batch, or always drain queue in one go). - Provide **test probe APIs**, such as an `ActorHandle` that exposes an async method to wait for the actor to reach a certain state, or a way to query the mailbox length, etc., for use in assertions.

These items will be clearly documented as part of Phase 05 deliverables. They improve the robustness and testability of the framework, checking off what Phase 04 couldn't do without an async foundation.

## Evolving the `statechart!` Macro for Async

The `statechart!` procedural macro will be updated to accommodate async actions, guards, and timed transitions. **Backward compatibility** is key: existing macro usage (defining purely sync statecharts) should remain supported with no changes. The macro will likely gain logic to detect `async` blocks or keywords inside state definitions, similar to Statig's approach where the presence of `async fn` triggers different codegen [5] [6].

**Async Actions:** In the macro syntax, we could allow specifying an action as `async { ... }` (for an entry/exit action or a transition effect). The macro will then generate an async function for that action and ensure it's awaited at the right time. For example:

```
state MyState {
    on Entry async {
        do_something_io().await;
    }
    on Exit { cleanup(); }
    on EventX => TargetState [guard] / async {
perform_async_side_effect().await; }
}
```

If the macro sees an `async` block for an action, it can generate the state handler as an `async fn` that awaits that block. The overall `StateMachine::handle(event)` may become async if any state requires it. We can leverage Rust's async/await within the generated code; the macro essentially wires up the Future chain. This is what Statig does: *"The macro automatically detects that async functions are being used and generates an async state machine."* [6] .

**Async Guards:** As discussed, directly awaiting in guards is tricky. We have a few options: - We could prohibit `await` in guard expressions and encourage the state pattern (this keeps determinism). In practice, that means if a user tries `on EventX [async { ... }]`, the macro could throw an error or warning. - Alternatively, allow it but clearly document that it will block the state transition until the future resolves, and no other events will be processed in the meantime (essentially treating it like a self-transition that completes later). This could be implemented by splitting the transition: the macro could generate a hidden state or use the internal event queue. For example, if you write an async guard that returns bool after awaiting, the macro might transform it into: spawn a background future that sets a flag and posts EventX again when ready. However, this complicates things and might confuse users.

Given best practices, we likely implement **timered and async decision logic via events** rather than true async boolean guards. The macro might support a syntax like `invoke async_fn(...) on outcome => StateY`, akin to XState's invoke pattern [10] . But as a first step, we will perhaps not allow arbitrary async in `[guard]` conditions.

**Timers in Macro:** We plan to extend the DSL to support time events. Syntax idea: `after(Duration) => TargetState / [action]`. The macro can transform `after(1s)` into setting up a timer on state entry. Implementation approach: - When entering a state with an `after(t) => S` transition, the generated code will start a timer (using `TimerService`) and store a handle or simply rely on an event when it fires. Possibly, we auto-generate an internal event variant like `Timeout_StateName` and the transition is encoded as `on Timeout_StateName => S`. - If the state exits before the timer fires (e.g. due to another event), we cancel the timer (cancellation of futures is done by dropping them; Embassy's timers can be canceled by dropping the `Timer` future).

The macro can hide these details. This will give a clear, high-level way to express timeouts.

**Statechart Macro Internals:** We will add new meta-items to the macro for enabling async. For example, we might introduce an optional attribute on the `statechart!` invocation like `async_handlers = true` or it might auto-detect. Statig's macro did auto-detection [6] which is convenient. With Rust's macro capabilities, we can inspect if any of the user-supplied handler code contains `await` or is an `async move || {}` block, etc. If so, we set a flag that switches the output. The output differences include: - The `handle(event)` function becomes `async fn handle(&mut self, event: E) -> Response` (where Response might be a future or requires await). - All state handler functions that were sync become `async fn` (if any one is async, probably all need to be async to satisfy trait impl coherence, unless we separate traits). - We might introduce an `AsyncStateMachine` trait if needed, or use the same trait but with different bounds under cfg.

**Macro for Actor Definition:** If we have a macro for defining an actor with an internal statechart, that macro also will incorporate async. E.g., `actor!` macro that wraps `statechart!` could output an async actor implementation.

To illustrate, suppose Phase 04 had:

```
statechart!{ TrafficLight:
    Initial => Red,
    state Red { on Timer => Green; },
    state Green { on Timer => Yellow; },
    state Yellow { on Timer => Red; }
}
```

Now in Phase 05, if we want the Timer event to be generated by a timer rather than external events, we could enhance:

```
state Red {
   on Entry { start_timer(30_sec); }  // macro generates a call to
TimerService::after
   on Timeout => Green;
}
```

The macro could generate that `start_timer` call in Red's entry action and a corresponding Timeout event. This shows how the macro evolves to orchestrate async behaviors seamlessly.

Finally, we ensure the macro's generated code remains `no_std` compatible. Any `.await` in generated code will compile in no_std as long as the futures in use are core futures (which they will be, using our TimerService trait for example). We only pull in executor specifics outside the statechart (in the runtime loop code).

The **bottom line**: the `statechart!` macro will be key to providing a **ergonomic API** for async features. Users should be able to declare an async action or a timed transition almost as easily as they declared sync ones, without worrying about the plumbing. Phase 05 will deliver an updated macro with these capabilities, guided by how Statig and XState handle similar needs (with our twist to maintain determinism and no_std compatibility).

## Phase 05 Technical Architecture & Implementation Plan

Bringing it all together, Phase 05's architecture extends the lit-bit framework in a modular, opt-in way. Here's an overview of the plan:

- **Core Architecture (No-Std):** The core actor and state machine logic remains in `no_std`, using zero-cost abstractions. We introduce new traits/structs (e.g. `AsyncActor`, `TimerService`) under cfg flags. The core message loop and statechart processing get generalized to handle async, but when compiled without async, they optimize down to the same old sync code (thanks to conditional compilation and inlining).

- **Async Integration Layer:** A new module/crate (or part of core behind feature) will contain integrations for:

- Tokio (feature `async-tokio`): includes Tokio `mpsc` for mailbox, spawns tasks on Tokio runtime, uses `tokio::time` for timers.
- Embassy (feature `async-embassy`): includes Embassy executor and channels, possibly provides a `#[embassy_executor::task]` annotation for actor tasks, uses `embassy_time` for timers. These might be two sub-features, or a single `async` feature that detects the target (std vs no_std) and activates the right parts (we may prefer explicit features for clarity).

- **Memory Model:** We design all data structures with fixed capacities (configurable via const generics perhaps). For example, `Mailbox<Msg, const N: usize>` might use a `heapless::Vec<Msg, N>`. We will test on real microcontrollers to ensure memory usage is predictable. Also, we document stack requirements of example actors to guide users.

- **Diagrams and Documentation:** We will prepare diagrams illustrating the runtime workflow (like the mailbox flow above) and the statechart structure. These will go into the developer docs. The architecture diagram might show how an event flows from an ISR or external source into an Embassy channel, then to an actor task, etc. Another diagram will depict the macro expanding a statechart with async internals.

- **Benchmark Suite:** Implement small programs to run on PC and on an MCU (e.g. an STM32 or an NRF chip in QEMU) to measure performance. Integrate those into CI using `cargo test` (with `--release` for accurate timing on hardware). Possibly use GitHub Actions with qemu or an embedded runner.

- **Testing Strategy:** Develop a new **async test harness** for unit-testing actors and statecharts. For example, on desktop we can use `tokio::test` attribute to write tests that await actor responses. On embedded, we might simulate events in a single-thread executor in test mode. We'll also include property-based tests (maybe using something like `proptest` for state machine tests) to ensure no deadlock or race when toggling async on/off. A special focus will be testing that the sync and async behaviors match when they should – e.g., a statechart run with artificial synchronous futures yields the same sequence as the purely sync version.

- **Task Breakdown & Timeline:** We propose a 3-week implementation broken down into roughly:

- *Week 1:* **Async Trait integration & Basic Tokio runtime integration.** Implement feature flags, update traits with default async support (ensuring no_std still compiles), and get a simple actor running on Tokio (e.g. an actor that prints messages with `tokio::spawn`). Also, implement the TimerService trait and get a basic timed transition working on Tokio (since `tokio::sleep` is straightforward). By end of week, have a demo of a statechart with an async action running in a Tokio `#[tokio::test]`.
- *Week 2:* **Embassy integration & no_std async.** This includes writing an example firmware (for e.g. an STM32) that runs an actor with async timers and messaging, using Embassy executor. Work out static allocation (perhaps require user to provide static buffers for now). Test on hardware or emulator. By mid-week, have Embassy example working (blinking an LED with a statechart, etc.). In parallel, enhance the `statechart!` macro to support the new syntax (async actions, after() events). By end of week 2, the macro changes should be mostly done, and all core tests (including new ones) passing in both sync and async mode.
- *Week 3:* **Deferred features & Polish.** Implement supervision mechanics: write a supervisor that can restart a failing actor (simulate a panic in an async handler and recover). Add message batching optimization (maybe allow a special mailbox type that delivers slices of messages). Implement test probes: e.g., an API to get a stream of state transition events from an actor for verification. This week is also for documentation: write extensive docs for the new features, including how to enable them, gotchas (like Send requirements), and performance implications. Finalize the benchmarking and have CI scripts in place to run them. By end of week 3, cut a Phase 05 release candidate with all tests passing and documentation updated.

Throughout the implementation, we will keep backward compatibility in mind – using CI to verify that examples from Phase 04 (without async feature) continue to compile and run unchanged. Performance will also be monitored as we introduce changes (hence the continuous benchmarking).

By the end of Phase 05, the `lit-bit` framework will support true asynchronous actors and statecharts with **zero-cost on embedded** and **full feature parity on std**. We will have unified the cloud and embedded worlds under one architecture, fulfilling the goals while keeping existing users happy.

**Sources:**

1. Ferrous Systems on no_std async: overhead of async_trait (heap allocation per call) [1].
2. Statig state machine with async (std-only via feature) [5] [6].
3. XState async guard pattern (invoke instead of direct async in guard) [10] [11].
4. Actix AtomicResponse for atomic async message handling [14] [15].
5. Maybe-async crate approach to dual async/sync API [20] [21].
6. Picoserve feature flags for tokio vs embassy integration [23] [25].
7. BackON crate's use of feature flags for timer backends (tokio, embassy, etc.) [26] [27].
8. Embassy executor: static allocation, no heap, timer queue built-in [43] [28].
9. Ractor framework supervision (linking actors, tests for supervision events) [41] [42].
10. Memfault on stack usage measurement via painting [38] [39].

[1] no_std async/await - soon on stable - Ferrous Systems
https://ferrous-systems.com/blog/stable-async-on-embedded/

[2] [3] [4] GitHub - tiannian/async-trait-static: Features like `async-trait`, avoid using `Box` and `dyn`.
https://github.com/tiannian/async-trait-static

[5] [6] [7] GitHub - mdeloof/statig: Hierarchical state machines for designing event-driven systems
https://github.com/mdeloof/statig

[8] [9] [10] [11] How to mix multiple guards and an "async" guard? · statelyai xstate · Discussion #3197 · GitHub
https://github.com/statelyai/xstate/discussions/3197

[12] [13] async_fsm — async Rust library // Lib.rs
https://lib.rs/crates/async_fsm

[14] [15] [16] [17] [18] [19] AtomicResponse in actix - Rust
https://docs.rs/actix/latest/actix/struct.AtomicResponse.html

[20] [21] [22] The bane of my existence: Supporting both async and sync code in Rust | NullDeref
https://nullderef.com/blog/rust-async-sync/

[23] [24] [25] picoserve 0.15.1 - Docs.rs
https://docs.rs/crate/picoserve/latest/features

[26] [27] Feature flags of BackON crate // Lib.rs
https://lib.rs/crates/backon/features

[28] [29] [30] [43] embassy_executor - Rust
https://docs.embassy.dev/embassy-executor/git/cortex-m/index.html

[31] embassy_executor - Rust - Docs.rs
https://docs.rs/embassy-executor

[32] Async fn in trait MVP comes to nightly | Inside Rust Blog
https://blog.rust-lang.org/inside-rust/2022/11/17/async-fn-in-trait-nightly.html

[33] C++ coroutines without heap allocation : r/rust - Reddit
https://www.reddit.com/r/rust/comments/1fy9t0u/c_coroutines_without_heap_allocation/

[34] Async fn in dyn traits without heap* is provably impossible
https://internals.rust-lang.org/t/async-fn-in-dyn-traits-without-heap-is-provably-impossible/15537

[35] Using DWT and other methods to count executed instructions on ...
https://developer.arm.com/documentation/ka001499/latest/

[36] Dwt in cortexm::dwt - Rust - kernel
https://docs.tockos.org/cortexm/dwt/struct.dwt

[37] riscv_simulator - crates.io: Rust Package Registry
https://crates.io/crates/riscv_simulator/1.0.1

[38] [39] [40] Measuring Stack Usage the Hard Way | Interrupt
https://interrupt.memfault.com/blog/measuring-stack-usage

[41] [42] ractor - Rust
https://docs.rs/ractor/latest/ractor/