ChatGPT

Cross-Runtime Actor Testing & Panic Recovery in Rust Actors

Async Test Infrastructure Across Tokio & Embassy

Cross-Platform Test Harness: Designing tests that run both on desktop (Tokio) and on embedded (Embassy) requires abstraction over the async runtime. A common pattern is to use conditional compilation to swap out runtime-specific code. For example, one can `#[cfg(test)]` use Tokio's timers in place of Embassy's, allowing the same async code to be tested on a PC. In one real-world case, an embedded button FSM used `#[cfg(test)]` to switch from `embassy-time` delays to equivalent Tokio delays for host testing [1]. Similarly, your **TestKit** can provide unified APIs that internally choose the appropriate runtime. In **lit-bit**, `TestKit` even has separate spawn methods for Tokio vs. Embassy (e.g. `spawn_actor_tokio` and `spawn_actor_embassy`) to ensure actors run in the correct executor [2]. This lets you write tests agnostic to the underlying runtime.

Zero-Heap Message Queues: Embedded targets often forbid dynamic allocation, so the actor mailboxes and test probes should avoid the heap. A proven strategy is to use **const generics** for fixed-capacity queues. In lit-bit tests, actors are spawned with a generic const size, e.g. `testkit.spawn_actor::<Calculator, 16>(calc)`, which likely uses a statically allocated ring buffer of 16 messages [3]. This approach (also used in frameworks like RTIC and Embassy) ensures no heap usage – if the queue is full, it can either block or drop messages as a back-pressure test scenario. Using structures like `heapless::spsc::Queue` or const-generic arrays for channels keeps the design zero-allocation. It's important in tests to simulate **mailbox overflow** for embedded back-pressure; your test plan includes checking mailbox overflow handling [4].

Async State Probes: To observe internal actor state changes without exposing them in production, **test probes** are invaluable. A probe can subscribe to actor lifecycle or state transition events. For example, lit-bit's `LifecycleProbe` hooks into an actor's `on_start` and `on_stop` callbacks. In tests, you spawn an actor with a probe:

```
let mut probe = LifecycleProbe::new();
let addr = testkit.spawn_actor_with_probe(my_actor, &mut probe);
// ... trigger actor startup ...
probe.wait_for_start().await;
```

Here, `wait_for_start()` asynchronously waits until the actor has finished its startup (or returns immediately if it already did) [5]. Under the hood, such a probe might be an `mpsc` channel or waker that the actor notifies on start/stop. Probes can also record errors: for instance, after a failed init you could check `probe.start_errors()` for any `ActorError` during startup [6]. This pattern lets tests **assert state transitions** (e.g. that an actor moved from "initializing" to "running" state) **without adding overhead**

in release. In lit-bit's design, `LifecycleProbe` and `SupervisionProbe` exist only in test or dev builds, and have minimal footprint in production.

Stream-Based Event Monitoring: Rather than polling a flag, probes can expose a **stream** of events. For example, an actor could emit a `StateChanged(new_state)` message into a `ProbeStream` (maybe a bounded channel) whenever it transitions. The probe's receiver implements `Stream`, so test code can do `if let Some(event) = probe_stream.next().await { ... }`. Using `StreamExt` utilities (like `timeout`, `take`, etc.) makes it ergonomic to wait for specific sequences of events. This approach is common in async testing: treating events as a stream means tests naturally await them in order. It's wise to keep the channel bounded (fixed size) to avoid unbounded heap usage. A small ring buffer (possibly using `heapless` on no_std) can store recent events. Notably, **Actix** includes a `Mocker` utility that can replace an actor with a test double emitting canned responses [7] – conceptually similar to a probe stream for messages. By using trait bounds or type aliases switched under `#[cfg(test)]`, one can swap the real actor for a dummy or attach a probe without affecting release builds [8].

Deterministic Async Testing Strategies

Controlled Time Advancement: Asynchronous code often involves timers, delays, and timeouts, which can make tests nondeterministic or slow. To ensure determinism, leverage **virtual time**. Tokio's built-in test utilities allow pausing time and advancing it programmatically. For example, setting `#[tokio::test(start_paused = true)]` will freeze the clock at start; then calling `tokio::time::sleep(duration).await` will *fast-forward* to advance time just enough to satisfy the delay [9]. This way, a 500ms sleep completes instantly in a test, ensuring consistent timing. You can also manually `pause()` and then `tokio::time::advance(delta).await` to simulate multiple timer events in sequence. Lit-bit's `TestKit` exposes similar control (`advance_time()`, `pause_time()` functions) [10], which likely tie into either Tokio's clock or an Embassy `VirtualTimer` for no_std. By only advancing time when no other future is ready, you get **deterministic ordering** of events [11]. The *tokio-rs* simulation framework takes this further, providing a **DeterministicRuntime** where *time advances only when the executor is idle*, forcing a reproducible task interleaving [11]. Such determinism is crucial for reproducible tests: you want the same sequence of actor message ordering every run. Consider using **loom** (the concurrency checking tool) if you need to explore many possible interleavings for race conditions, but for most FSM scenarios, a single-threaded deterministic scheduler is sufficient.

Timeouts & Cancellation in Tests: It's good practice to wrap test futures in a timeout to avoid hangs. For instance, using `tokio::time::timeout(duration, some_future).await` will fail the test if `some_future` doesn't complete in the expected time. This ensures that if an actor deadlocks or a message never arrives, the test doesn't stall indefinitely. In your test harness, you might integrate this by default (e.g., `TestKit::advance_time()` could also fail if the system hasn't quiesced by a certain simulated time). Another pattern is to inject **cancellation signals**: for instance, sending a special message to gracefully stop actors after the test scenario, then awaiting their termination. Lit-bit's `shutdown_actor(addr).await` call in tests exemplifies this – it stops the actor and then you can `probe.wait_for_stop().await` to confirm cleanup [12]. Deterministic testing means also controlling *when* and *in what order* actors are stopped, so your harness should coordinate shutdown sequence (as indicated by the "graceful shutdown sequences" item in the checklist [13]).

Property-Based FSM Testing: For complex state machines, **property testing** can systematically validate invariants. Using frameworks like *Proptest* or *QuickCheck*, you can generate random sequences of input messages and ensure the actor's observed state or outputs meet certain properties. The `proptest-state-machine` crate specifically supports modeling an abstract state machine and comparing it against the implementation [14] [15]. For example, you could model a simplified version of your actor's state transitions (perhaps ignoring timing) and have proptest drive both the model and the real actor with the same sequence of events, then assert that key properties hold (e.g., "if a Reset message was sent, the actor's value must be 0 afterward"). The framework will try to find minimal counter-examples if a property is violated [14] – effectively giving you small reproducible test cases for complex bugs. While property tests might not run on embedded directly, you can target the model against the sync or Tokio version of your actor. Ensure determinism here by seeding any randomness (Proptest does this by default with a seed that can reproduce the failing case [15]). This approach is excellent for finding edge-case sequences that break assumptions, especially in concurrency or order-of-operations.

Isolated Actor Simulation: To test an actor in isolation, provide it with **mock dependencies** and run it in a lightweight harness. This often means using a dummy implementation of external traits or hardware. For instance, if an actor uses an `embedded-hal` trait to talk to a device, you can use `embedded-hal-mock` to simulate device behavior in a no_std environment [16]. In the forum example, the developer mocked a digital input and controlled the timing of button presses to test a debounce state machine entirely in Tokio [17]. Your TestKit likely offers utilities like `MockActor` or `MessageCapture` [18] [19] that let you stub out an actor's collaborators or intercept messages it sends. For example, `MessageCapture<M>` can record all messages of type M sent to a channel, allowing the test to later assert that "message X was sent after event Y" [19]. This is akin to how integration tests can capture logs or outputs, but here at the actor-message level. By simulating the environment (e.g., using `TestKit::set_env(key, value)` to provide config or environment inputs in tests [20]), you can force error paths (as shown in the `test_initialization_failure` where an invalid DB URL is set to make `on_start` fail [21] [22]). The goal is **deterministic, isolated reproducibility** – each test sets up the exact initial state and uses probes to observe exactly what the actor does.

Panic Recovery & Supervision in Async Actors

Catching Panics in Async Contexts: In a robust actor system, one misbehaving actor shouldn't bring down the entire application. Rust panics are unwinds (unless set to abort) that can be *caught* at boundaries. However, catching panics in async code is tricky because Rust lacks a stable `async catch_unwind`. The common workaround is to spawn the actor's event loop as a task and catch panics at the task boundary. Tokio by default **catches panics in spawned tasks** – the panic doesn't crash the runtime thread; instead, it marks the task's `JoinHandle` as failed (you can inspect `JoinHandle.await` for `Err(JoinError)` which indicates a panic). In fact, Tokio's default for unhandled task panics is to **ignore** them (they're just logged) [23], meaning the rest of the system keeps running. This is similar to "let it crash" philosophy: the actor's task ends, but everything else continues. To handle this, your supervisor can await the JoinHandle and detect the panic.

Under the hood, one can use `std::panic::catch_unwind` with an `AssertUnwindSafe` wrapper to catch a panic within a Future. This was the pattern in early futures (`Future::catch_unwind` in `futures` 0.1) which required the future to be `UnwindSafe`. In practice, it means ensuring the actor's state can be safely reused after a panic – or more commonly, you discard the actor instance and start a new

one. The Rust documentation emphasizes that using `catch_unwind` for logical control flow is unusual and should **not** be a substitute for proper error handling; its primary valid use-case is isolating faults so the process can continue running [24]. In an actor system, that's exactly our intent: if one actor panics, catch it so the rest survive, then decide how to recover. Do note that if your project sets `panic = 'abort'` (common in no_std for smaller binary or to avoid unwind cost), **no panic can be caught** – the process (or device) will reset immediately [25] [26]. For embedded, you might keep unwind panics enabled during testing, but in production firmware often abort is used for simplicity. Ractor's documentation explicitly calls this out: with `panic = 'abort'`, panics won't enter the supervision flow [25].

Supervision and Restart Strategy: Building on panic catching, a supervisor can implement various restart policies. The simplest is **always restart**: when a child actor panics (or otherwise terminates unexpectedly), the supervisor creates a fresh instance of that actor. The Actix framework historically did not automatically restart actors on panic – in fact, it had *no default monitor/restart strategy*, and a panic in an Actix actor could just kill the thread or hang the Arbiter [27]. Actix developers are encouraged to handle errors as `Result` instead of panicking [28]. In contrast, **Ractor** provides a full supervision tree: a parent is *notified* if a child dies or panics, via a `SupervisionEvent` message [29]. The parent can then choose to restart the actor. Ractor's design is very much like Erlang's OTP – children link to their supervisors and report their termination or fault reasons [30]. In lit-bit, a similar concept is planned: the `SupervisionProbe` in your test API can `wait_for_panic()` and `wait_for_restart()` events [31], meaning the framework will catch the panic and trigger a supervised restart (with details like `last_restart_strategy` available for inspection in tests). This suggests an implementation where the actor runner catches panics, possibly logs or records the error, then uses a predefined `RestartStrategy` (maybe immediate restart, exponential backoff, etc.) to decide what to do next. The supervision logic should also cap how many times to restart to avoid infinite crash loops (a common pattern is "restart up to N times within M seconds, otherwise give up").

Resource Cleanup After Panics: One challenge with a panic is cleaning up what the actor was doing. Fortunately, if we unwind, all Rust destructors for that actor's stack are called, so things like `&mut` guards, heap allocations owned by the actor, etc., will drop. However, an actor might hold external resources: open file handles, network sockets, or perhaps child actors it spawned. A robust system should handle those. One approach is to have the supervisor take responsibility: when it sees a child failed, it could, for instance, drop the channel to that child (freeing buffers) and spawn a new child with a fresh channel. If the actor was managing a hardware resource, you might need to reset that hardware or reclaim it. In lit-bit, the `on_stop` callback is meant for cleanup on normal shutdown – but in a panic scenario, `on_stop` may not run automatically. You could consider calling an actor's `on_stop` in the catch_unwind handler **before** dropping the actor, to give it a chance to release resources (if it's safe to do so). Ractor's docs note that panics in certain stages (like `pre_start`) don't send SupervisionEvents because the actor isn't fully alive yet [32] – in those cases, the framework just fails the spawn. But panics during message handling (`handle`) or `post_stop` do notify the supervisor [32]. This means even during a panic-triggered shutdown, you might get a chance to run some cleanup logic in the supervisor (or in the actor's overridden `restarting` method, as Actix provides [33] where an actor can customize what to do before a restart).

Example – "Let it Crash" Resilience: To illustrate, imagine an actor that monitors a sensor and occasionally panics (perhaps due to an internal bug). With supervision, the panic is caught and converted into a message to the supervisor: *"child X panicked"*. The supervisor (which could be the top-level system or a parent actor) receives this and decides to restart that sensor actor. It could log the incident, and then call something like `start_new_sensor_actor()`. The new actor instance might reinitialize the sensor (power-cycle it, etc.) as part of its `on_start`. Meanwhile, any messages that were in the old actor's mailbox might be lost

(depending on design), but the system as a whole keeps running. This is analogous to isolating faults: one actor crashing doesn't cascade into others. When writing tests for this, you can deliberately inject a panic in an actor (perhaps by sending it a special test message that causes `panic!()`), and then use `SupervisionProbe.wait_for_panic().await` to verify the supervisor caught it, and `wait_for_restart()` to ensure a new actor was started in its place [31]. This validates that your **panic recovery strategy works as designed**.

Comparing Actor Frameworks & Probe Performance

Actor Frameworks Comparison: Rust's ecosystem has numerous actor libraries, each with different emphases. **Actix** is one of the oldest and focuses on ergonomics and integration with Actix-Web. It uses a threadpool under the hood. Testing Actix actors usually involves spinning up a System and using the actor's address to send messages. Actix doesn't have special test probes; you rely on normal message responses or state queries. As mentioned, Actix's approach to panics is essentially *avoid them* – it doesn't restart actors automatically [27]. Developers can use the `Supervisor` utility to wrap an actor; Actix's `Supervisor::start()` will respawn an actor if it stops, but only if the actor's `Context::stop` was called – a panic might not be cleanly caught. In contrast, **Ractor** (a newer framework inspired by Erlang) has first-class support for supervision and clustering. Ractor exposes a `SupervisionEvent` message type and will send events for *"unhandled panic!"* to the parent [29]. This makes it easier to test – you can have a test supervisor actor that records if it got a panic event from a child. On async testing, Ractor doesn't impose a specific runtime (it works with Tokio under the hood, using `#[tokio::main]` in examples [34]). You would test it similarly: spawn actors and await their `JoinHandle`s or use channels to verify interactions. Another framework, **xtra**, embraces purely async/Await without a separate actor context type – it's simple to use with Tokio and you test it by just `.send()`ing and `.await`ing responses (very similar to what lit-bit aims for). However, xtra also doesn't automatically recover from panics (if an actor future panics, that task ends).

It's worth noting that beyond Actix and Ractor, many actor libraries exist (Kompanion, bastion, or even higher-level like **Kompact** in Rust). Each has their testing tricks. For example, Kompact (inspired by Akka) provides a testing API where you can schedule events and use "probe" actors that expect certain messages. The themes are common though: controlling execution, observing messages, and isolating failures.

Low-Overhead Test Probes: Instrumentation in tests should impose near-zero overhead in production. Strategies to achieve this include **conditional compilation** and **feature flags** for test code. As shown earlier, one can use `#[cfg(test)]` or a dedicated feature (e.g. `features = ["test-probes"]`) to compile in extra fields or hooks only when running tests. For instance, an actor could have an optional `probe: Option<ProbeChannel>` that is only present in test builds. This way, in release builds the compiler can optimize out any probe-related code entirely. The code snippet from an Actix article demonstrates this pattern by type aliasing to a `Mocker` when testing [8]. In lit-bit, the `test_utils` module (TestKit, probes, etc.) is likely behind a `cfg(test)` or non-default feature so that normal builds don't even include it. This ensures **zero cost** unless you explicitly enable the test support. Another technique is to use macros: e.g. a `probe!(event)` macro that expands to nothing in non-test builds, but in test builds logs the event to a global or sends to a channel.

When measuring performance (benchmarks), you should run on a build with probes **disabled**, to get realistic throughput and latency numbers. Otherwise, the instrumentation (which might use locks, channels, or extra branches) could skew results. It's common to have separate Cargo features for "bench" vs

"test" vs "production". For example, you might enable a `logging` or `tracing` feature during development and testing, but turn it off for release/bench to see the raw performance. If you do need to measure the overhead of your probes, keep those benchmarks separate (so you can quantify, say, "the probe slows actor throughput by 5% under load"). Lit-bit's performance test kit likely uses minimal instrumentation, focusing instead on measuring message rates and latency with high precision timers [35]. They even have a `LatencyMeter` for stats collection [36]. Such measurement code should be careful to minimize perturbation – e.g., record timestamps with a lightweight call (perhaps using `Instant::now()` which is cheap on std, or a cycle counter on embedded). And if it stores data, use a pre-allocated buffer to avoid allocator interference during the test.

Benchmarks Isolation: When writing async benchmarks (using `criterion` or libtest harness), isolate them from test scaffolding. For instance, do not reuse a global TestKit with probes while benchmarking actor throughput – create a fresh minimal setup. This avoids any lingering state from tests affecting your benchmark (like a probe consuming CPU or memory). It's often useful to provide *benchmark-specific harnesses* – perhaps a stripped-down executor or using the `PerformanceTestKit` you designed. The `PerformanceTestKit.wait_for_quiescence()` [37] suggests it can wait until the system is idle before taking measurements, which is great for consistent results. Also consider pinning the runtime to a single thread for more predictable performance in benchmarks (reducing variance). In summary, treat benchmark runs as a different "profile" of your system: minimal observers, full optimizations, and sometimes even different compile-time settings (e.g., using `Cargo.toml` `[profile.bench]` to ensure it's optimized like release). This separation ensures that your **test probes deliver value in development** but **vanish in production**, giving you confidence that the zero-cost abstraction promise holds.

Conclusion

By integrating these patterns – cross-runtime test harnesses, async probes, deterministic scheduling, and robust supervision – the `lit-bit` library can achieve reliable and portable tests. You can simulate complex actor interactions with confidence that tests will produce the same outcomes on PC and on embedded targets. Adopting proven approaches from other actor frameworks (Actix's test organization, Ractor's supervision model, Erlang's "let it crash" philosophy, etc.) will strengthen the library. Crucially, all these testing and supervision capabilities can be included without sacrificing runtime performance by leveraging Rust's zero-cost abstractions and conditional compilation. The result will be an actor system that's not only efficient in production, but also thoroughly verified across a spectrum of scenarios – from basic arithmetic actors to fault-injection chaos tests. With async actors, **deterministic testing and resilient recovery** go hand-in-hand: your project's emphasis on both will lead to a more robust and maintainable system overall.

Sources:

- Tokio async testing techniques (time pausing and simulation) [9] [11]
- Rust forum example of Embassy vs Tokio in testing (cfg switches and mocks) [1] [16]
- Lit-bit internal testing guide (TestKit, probes, examples of usage) [3] [5]
- Proptest state machine testing for verifying FSM properties [14]
- Rust panic catching in async (best practices and caveats) [24] [25]
- Actor frameworks comparison: Actix (no auto-restart) [27] vs Ractor (supervision events on panic) [38] and testing patterns [8].

[1] [16] [17] How to test this async logic? - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/how-to-test-this-async-logic/103272

[2] [3] [4] [5] [6] [10] [12] [13] [18] [19] [20] [21] [22] [31] [35] [36] [37] test-guide.md
https://github.com/0xjcf/lit-bit/blob/2b12f648efc4d07f7678695d16590156e6a454df/docs/test-guide.md

[7] [8] [33] Mocking Actix Actor without getting a gray hair
https://www.amarjanica.com/mocking-actix-actor-without-getting-a-gray-hair/

[9] Unit Testing | Tokio - An asynchronous Rust runtime
https://tokio.rs/tokio/topics/testing

[11] GitHub - tokio-rs/simulation: Framework for simulating distributed applications
https://github.com/tokio-rs/simulation

[14] [15] State Machine testing - Proptest
https://proptest-rs.github.io/proptest/proptest/state-machine.html

[23] Builder in tokio::runtime - Rust - Docs.rs
https://docs.rs/tokio/latest/tokio/runtime/struct.Builder.html

[24] [26] Panics - Comprehensive Rust
https://google.github.io/comprehensive-rust/error-handling/panics.html

[25] [29] [30] [32] [38] ractor - Rust
https://docs.rs/ractor/latest/ractor/

[27] [28] The Hitchhiker's Guide to The Actor Model in Rust
https://ryankung.github.io/pdfs/2019-01-25-guide-to-actor-mode-in-rust.pdf

[34] GitHub - slawlor/ractor: Rust actor framework
https://github.com/slawlor/ractor