ChatGPT

Panic Recovery and Supervision in Async Actor Frameworks

Production-Ready Solutions: Panic Detection & Recovery

Actix: The Actix actor framework provides a built-in `Supervisor` mechanism to restart actors that stop unexpectedly. An actor can implement the `Supervised` trait, defining a `restarting()` hook to reset state on restart [1]. If a supervised actor's event loop *fails* (e.g. panics or otherwise stops), Actix's `Supervisor` creates a new execution context and invokes the actor's `restarting` method before resuming message processing [1]. Actix does not explicitly propagate panic details to the supervisor – a panic in an actor typically results in that actor's thread (Arbiter) stopping. By default, Actix treats panics as fatal (the developer often enables `panic = "abort"` in production) or relies on the OS thread isolation. The emphasis is on preventing panics; however, the supervisor will automatically restart a crashed actor (one-for-one strategy) if it was started via `Supervisor::start` [2]. This provides basic fault tolerance, but without rich error reporting – the actor just restarts fresh, and the cause of failure isn't delivered to user code (apart from logging). Actix's design favors simplicity and performance: there is minimal runtime overhead for supervision until an actor actually fails, since messages are handled normally and supervision only kicks in on a *stop* or panic event.

Ractor: Ractor is heavily inspired by Erlang's OTP model and includes a full supervision tree. Actors are *linked* to supervisors such that any abnormal termination sends a **supervision event** to the parent. Internally, Ractor catches actor panics and errors and converts them into a `SupervisionEvent::ActorFailed` message containing the panic info or error as an `ActorProcessingErr` [3] [4]. For example, if an actor panics (when using `panic = "unwind"`), Ractor's runtime captures the panic payload (using Rust's unwind mechanism) and notifies the supervisor with an `ActorFailed` event (including the panic as an error) [5]. The supervisor (which is usually itself an actor) can then decide how to respond – e.g. log the failure, attempt a restart, or escalate it. Ractor differentiates *normal exits* from *failures*: a child that stops cleanly sends `ActorTerminated` (with an optional exit reason), whereas an unhandled panic or error triggers `ActorFailed` with the error attached [3] [4]. This allows supervisors to implement policies like "only restart on abnormal exit." Ractor does not automatically restart actors by default; instead, it provides the primitives (linking and events) for the user or a higher-level library to implement restart strategies. Indeed, the community crate **ractor-supervisor** builds on these events to offer OTP-style strategies (OneForOne, OneForAll, RestForOne) and to handle *meltdowns* (too many restarts in a time window) [6] [7]. In terms of panic info, Ractor captures the panic payload (downcasting to retrieve the message if it's a `&str` or `String`) and stores it in the `ActorProcessingErr` (a `Box<dyn Error>`). This means a supervisor can access a string description of the panic (though not a full stack trace) via the error object. The use of message-based failure signals keeps the design **deterministic** – failures are handled in the actor's message queue order – and incurs little overhead during normal operation (the actor's `handle` returns a `Result`, which is `Ok(())` in the common case, and only on error or panic does it produce an Err) [8].

Bastion: Bastion is a fault-tolerant async runtime that bakes in supervision at its core. It organizes actors into supervised *children groups* and supervisors can oversee other supervisors, forming a hierarchy. Bastion implements **OTP strategies** out-of-the-box: by default the root supervisor uses one-for-one, and you can configure supervisors to use one-for-all or rest-for-one as needed [7]. If any actor in a group fails (panics or returns an error), Bastion's supervisor will restart actors according to the configured strategy [7]. For example, under *OneForAll*, if one child panics, *all* siblings are stopped and restarted; under *RestForOne*, the panicking actor and any that were started after it will be restarted [7]. Bastion detects panics by running each actor in an isolated lightweight process (using the `lightproc` executor) – effectively, each actor's async task is guarded. If a task ends with an error or unwinds, the Bastion framework catches that. The actor's `exec` closure returns a `Result<(), ()>` where returning `Err(())` (or a panic leading to an Err) signals failure [9] [10]. The supervisor then automatically performs the configured recovery: it can restart immediately or apply a back-off strategy. **Restart strategies** in Bastion are flexible – you can set a policy like *Always* restart, *Never* restart, or restart up to N *Tries*, as well as use timing strategies (immediate, linear backoff, exponential backoff) to avoid tight restart loops [11] [12]. Bastion does not forward a full stack trace of a panic to user code, but it does log the panic. It typically treats any panic as an actor failure equivalent to an `Err(())` from the actor's `exec`. The emphasis is on automated recovery rather than inspection of the panic – the supervisor knows *which* child failed and when, but not necessarily the detailed cause. However, since Bastion gives the developer control to define "when a certain condition is met" to kill or restart actors [13], one could implement custom logging or error-handling in the actor's code (or via custom callbacks) to capture more info. Performance-wise, Bastion's model adds a small overhead for supervision bookkeeping (tracking children and their states), but message passing and actor execution remain asynchronous and scalable. The supervision checks (e.g. incrementing restart counters, scheduling a restart) occur only on failures, so the steady-state cost is low – making it compatible with **zero-cost** aspirations when no faults occur.

Unified Abstractions Across Tokio and Embassy

To support both the *Tokio (std)* and *Embassy (no_std)* environments, a unified abstraction is needed so that panic/failure recovery works uniformly. The `lit-bit` library approaches this by introducing common types and traits to represent supervisor actions and failure events in a platform-agnostic way. For example, it defines an `ActorError` enum and a `SupervisorMessage` that encapsulate child status updates [14] [15]. A child that stops normally or is shut down can be represented by `ChildStopped`, whereas a panic or runtime error triggers a `ChildPanicked` message with the child's ID [15]. Both Tokio and Embassy implementations can create or handle these messages, ensuring the **same supervision logic** runs on both platforms. The design goal is to treat a panic just like any other message to the supervisor – thereby maintaining deterministic ordering of events. In fact, one of the design principles is that *failure notifications are processed as regular messages* in the statechart's event loop [16]. This means a panic in a child actor results in an event that the supervisor (which itself can be an actor/statechart) handles in-turn, rather than an out-of-band interruption. Such an approach mirrors Ractor's use of supervision events and gives consistent behavior across platforms.

Capturing Panic as Errors: In a Tokio context (with `std` available), the library can use `std::panic::catch_unwind` or tokio `JoinHandle` introspection to catch panics without crashing the whole process. For example, Tokio's `JoinHandle<Result<(), ActorError>>` for a child actor can be polled: if it finishes, we check if it returned an `Err` or panicked. A panic is indicated by a `JoinError` (`is_panic()`), which can be converted into an `ActorError::Panic` for that actor [17]. Additionally, the panic's payload can be downcast to retrieve a message – many frameworks simply capture the panic as a

string. In fact, the lit-bit example code uses exactly this approach: when a worker actor's handler catches an unexpected panic, it downcasts the payload to `&str` or `String` to get the panic message (or uses a default message) [18]. This string could then be logged or even sent to the supervisor as part of a failure report. By abstracting this behind an `ActorError` or similar, the *type* of panic (e.g. the panic message or type of payload) is preserved as a variant, and the supervisor can handle it uniformly (e.g. decide to restart the actor and perhaps print the error message).

On the Embassy side (no_std), unwinding is not available by default – panics will invoke the panic handler (and typically abort). To provide a similar abstraction, lit-bit uses a strategy of *simulating* panic catches via its API. Instead of literally catching an unwind (which you cannot do in stable no_std), the framework encourages actors to return errors in controlled failure situations, which are treated like panics by the supervisor logic. For instance, an Embassy-based actor might call a special failure method or simply return an `Err` from its `handle` future to signify it wants to "crash." The supervisor can be notified via a `SupervisorMessage::ChildPanicked` in that case, just as if a real panic occurred. Under the hood, lit-bit's Embassy integration could use a lightweight signal: since there is no `JoinHandle` to poll, one approach is to have the child actor's task send a message to the supervisor when it terminates. For example, the actor's future can be wrapped such that on completion (or on a drop), it sends a `ChildStopped` or `ChildPanicked` signal. This is hinted by the design: the supervisor struct in lit-bit keeps a list of children and provides methods like `poll_children` (for Tokio) and likely analogous checks or callbacks for Embassy. Notably, lit-bit uses conditional compilation to use a **HashMap** of children on std vs. a fixed-capacity **heapless map** on no_std [19], but provides the same API. Similarly, a `RestartStrategy` enum (OneForOne, OneForAll, RestForOne) is defined generically and used for both runtimes [20]. The supervisor logic (deciding which actors to restart) can thus be written once, using the child list and strategy, and it will apply equally whether children are Tokio tasks or Embassy tasks.

Communicating Panic Events: A key part of unifying the behavior is having a common *message format* for "actor failed" events. In lit-bit, the `SupervisorMessage<ChildId>` includes variants like `ChildPanicked { id: ChildId }` to convey that a given child actor has panicked or failed [15]. In a Tokio environment, the library can generate this message when a JoinHandle completes with an error or panic. In an Embassy environment, because there's no threading, the child actor itself (or a small wrapper) can enqueue the same message (for example, by sending into a channel that the supervisor is polling) right before it terminates (if it detects an error). This abstraction ensures the *supervisor actor* doesn't need to know about join handles or specific Embassy mechanisms – it only deals with `SupervisorMessage`s. The result is that a supervisor can implement an `on_child_failure(id)` callback (as defined by the `Supervisor` trait in lit-bit) and decide a restart strategy in a platform-agnostic way [21] [22]. The framework then applies that strategy: e.g. for OneForOne, it will respawn the one child (using a stored restart factory closure for Tokio [23] [24], or by calling the child's constructor again in no_std), for OneForAll it iterates over all child IDs, etc. All of this can be done uniformly because the events and strategy enumeration are the same on both platforms.

To handle *panic information* uniformly, one practical approach is to limit the information to what's available in no_std. Since on embedded targets a panic will often just yield a static message (if any), lit-bit can standardize on carrying a string or static `&'static str` as the panic description in `ActorError`. Indeed, its `ActorError::Panic` variant could optionally hold a message. In std mode, this message is filled with the downcast panic payload (so you might see *"index out of bounds"* or whatever panic string), whereas in no_std mode it might be a fixed string like *"panic occurred"*. This way, the supervisor can at least log the event. More advanced use could involve capturing a backtrace on std (using

`std::backtrace::Backtrace` in the panic hook) – some systems do this outside the actor system (e.g. a panic hook writing to a log). However, for determinism and simplicity, frameworks often avoid trying to capture full stack traces programmatically and instead focus on structured error reporting (the supervisor knows which actor failed and maybe a reason string).

No_std Embedded Strategies for Panic Handling

In an embedded async runtime like Embassy (running without an OS or standard lib), true runtime **panic recovery** is challenging, because the default panic behavior is to halt or reset the system. There are, however, strategies to make the system resilient and to test those failure scenarios:

- **Panic Hooks and Fault Handlers:** In no_std, you can define a custom `#[panic_handler]` to run when a panic occurs. This can, for example, log the `PanicInfo` (which includes the file, line, and optional message) via an LED or serial output, and then trigger a system reset. While this doesn't *resume* execution, it does allow the system to fail fast and be restarted (which, at a whole-device level, is analogous to a one-for-all supervisor restart). Some embedded frameworks set up a watchdog timer such that if a panic occurs and the system halts, the watchdog will reset the microcontroller, achieving a form of automatic recovery. The downside is that all state is lost (all actors are restarted en masse). If a finer-grained recovery is needed (e.g. restart one subsystem task but keep others running), it requires more complex setups like an RTOS.

- **Simulating Panics in Tests:** Because we can't unwind on embedded targets in stable Rust, one approach is to simulate actor panics for verification. Lit-bit can be tested in a desktop environment by enabling the Embassy support but using `panic = "unwind"` during tests. For example, one could run an Embassy-based statechart in a special test harness where a child actor deliberately calls `panic!()` or uses an assertion failure. On a desktop (std) target, this panic can be caught by the test harness or by `catch_unwind` around the executor, allowing the supervisor logic to be invoked. This way, you can ensure that your `SupervisorActor` would schedule a restart. Another technique is to provide a testing feature where the actor's code uses a macro like `debug_panic!()` that in std mode does `panic!()` but in no_std might simply return an error. Using such conditional compilation, you can force error paths that mimic panics without actually invoking the panic handler on device. This enables unit/integration tests of the supervision logic – e.g. asserting that after a simulated "panic" event, the supervisor's children list is updated and the actor is restarted with a fresh mailbox, etc.

- **Resource Cleanup:** When an actor "fails" on embedded, we must ensure resources are cleaned up to avoid leaks or deadlocks. In a no_std environment with no operating system, resources are often statically allocated (e.g. using `heapless` queues, hardware peripherals). If an actor is removed or restarted, the supervisor should reclaim or reinitialize these resources. Lit-bit's design using a fixed child slot for each actor (e.g. an array or `FnvIndexMap` of children) helps here [25]. When a child is marked failed, the supervisor can drop its `Address` (which closes the message channel) and remove it from the table. Any tasks still attempting to send to that actor's inbox will receive an error (since the inbox is closed), preventing further queuing on a dead actor. Before restarting that actor, the supervisor can create a fresh channel/mailbox for the new instance. This effectively discards any messages that were in the old queue – similar to how OTP treats a mailbox of a terminated process. Timers and other async resources in Embassy are usually tied to the task's lifetime. For example, a

delay (`Timer::after`) future will be canceled automatically if the task is dropped (because the future is dropped). Thus, if an actor panics and the whole device resets, all peripherals restart cleanly; if we were hypothetically able to drop just that task, Embassy's executor would free any pending awaits for that task. In practice, without unwinding, we rely on the full reset, but during testing (with unwinding), it's worth verifying that after an actor's failure, no lingering timer interrupts or DMA transfers continue. One can simulate this by having the actor acquire a resource (like start a one-shot timer or toggle a GPIO) and then panic – after recovery, the new actor should be able to acquire or re-init the resource (the supervisor might need to reconfigure hardware or ensure the previous instance's state is overwritten).

- **No_Std Restart Mechanisms:** If using an RTOS or a soft-failure approach, one might implement a custom scheduler that catches panics at the *thread* level. For instance, an RTOS task wrapper could catch a panic from a Rust thread (using `setjmp/longjmp` in C or similar, which is not idiomatic nor entirely safe in Rust) and then signal a supervisor task. This is complex and usually avoided due to Rust's lack of officially supported unwinding on bare metal. Instead, embedded developers lean on fail-safe designs: isolate critical components into separate hardware (or software) contexts when possible. For example, if using a dual-core MCU, one core could supervise the other. Absent that, the system reset is the fallback.

In summary, **embedded panic handling** is more about designing for reset and using supervisor patterns at a coarse grain. Lit-bit's aim is to *encapsulate the panic recovery behind the same API for both desktop and embedded*. That means the supervisor logic doesn't change – it calls `on_child_failure` and applies a strategy whether running on std or no_std. On no_std, `on_child_failure` might simply always choose to restart the actor (since there's no OS to kill the whole process). The restart will re-initialize the actor's state chart deterministically. The framework ensures *deterministic execution* by controlling when restarts happen (for example, maybe delaying them until a safe point or processing them in order). And it upholds **zero-cost** principles by avoiding any runtime cost when no failure occurs: in no_std mode, there are no heap allocations for supervision (fixed buffers are used) and no continuously running background checker – instead, failures surface as messages or via a lightweight poll. In Tokio mode, the overhead is also low: the use of `JoinHandle` polling and `catch_unwind` is pay-for-what-you-use (the cost of setting up unwind catching is minimal, and only when a panic occurs does it incur the stack unwinding cost).

Finally, both environments benefit from **test strategies** that ensure robustness: exhaustive testing of statechart actors under simulated failure conditions (like injecting faults) can reveal any nondeterministic behavior or resource leakage. By studying Actix, Ractor, and Bastion, the lit-bit project can combine their best practices – Actix's simple restart hooks, Ractor's rich failure messaging, and Bastion's flexible strategies – to implement a robust panic recovery system. The end result will be an async statechart runtime that can automatically recover from actor panics with OTP-style supervision on desktop and embedded targets alike, without sacrificing performance or determinism in the common case of no failures.

Sources:

- Actix actor supervision and restart behavior [1]
- Ractor's supervision events for actor start/stop/failure [3] [4] and documentation on catching panics [5]
- Bastion's one-for-one, one-for-all, rest-for-one strategies [7] and configurable restart policies [11]

- Lit-bit design notes on platform-agnostic supervision (zero-allocation, deterministic messages) [16] and example of capturing panic info via downcasting [18]

---

[1] Supervised in actix - Rust
https://docs.rs/actix/latest/actix/trait.Supervised.html

[2] rust - Why does my actix Supervisor not retry stopped actors? - Stack Overflow
https://stackoverflow.com/questions/50526516/why-does-my-actix-supervisor-not-retry-stopped-actors

[3] [4] messages.rs
https://github.com/slawlor/ractor/blob/38faea5de8347f6a02fdea45d7480625ab3b5f79/ractor/src/actor/messages.rs

[5] [8] ractor - Rust
https://docs.rs/ractor/latest/ractor/

[6] [7] ractor_supervisor - Rust
https://docs.rs/ractor-supervisor/latest/ractor_supervisor/

[9] [10] supervisor.rs - source
https://docs.rs/bastion/latest/src/bastion/supervisor.rs.html

[11] [12] RestartStrategy in bastion::supervisor - Rust
https://docs.rs/bastion/latest/bastion/supervisor/struct.RestartStrategy.html

[13] Bastion
https://www.bastion-rs.com/

[14] [15] [16] [20] [21] [22] mod.rs
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/lit-bit-core/src/actor/mod.rs

[17] [19] [23] [24] [25] supervision.rs
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/lit-bit-core/src/actor/supervision.rs

[18] supervision_and_batching.rs
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/lit-bit-core/examples/supervision_and_batching.rs