ChatGPT

Async Supervision and Message Batching in a `no_std` Dual-Runtime Actor System

1. Async Supervision in `no_std` (Embassy) vs. `std` (Tokio)

Challenges: In a `no_std` embedded context, we lack OS threads and rich dynamic features like `futures::select_all` for monitoring multiple tasks. The goal is to supervise child actors (tasks) such that if one *panics* or stops unexpectedly, the parent (supervisor) is **notified and can react**. This must work under Embassy's cooperatively scheduled executor (no dynamic allocation, often `panic = abort`) and also under Tokio's threaded runtime.

- **Embassy (Embedded) Patterns:** Embassy tasks are statically allocated and typically run forever or until explicitly stopped. There's no built-in `JoinHandle` or task cancellation API – if a task panics under `panic = abort`, the whole system halts. Thus, **preventing uncontrolled panics is crucial**. Instead of relying on catching panics at runtime (impossible with abort-on-panic [1]), each actor should handle errors internally and signal failure to its supervisor via messages. For example, a child actor can send `SupervisorCommand::ChildFailed(id)` to its parent before gracefully stopping. This requires coding the actor's loop to catch errors (as `Result`s or via internal checks) rather than truly panicking. Embassy's channels can help here: since an Embassy channel **never closes** (receiver lives "forever"), a blocked sender on a dead task would deadlock [2] [3]. To avoid this, the supervisor should drop a child's `Address` when it's done to prevent further sends. Task **cancellation** must also be cooperative: for example, share an `AtomicBool` or an `embassy_sync::signal::Signal` that the child checks periodically to exit its loop. In summary, embedded supervision uses **static lists of children, error signaling via messages, and cooperative cancellation** (no `select_all`). This static design is deterministic and panic-safe: if a child stops or signals an error, the parent's next `on_event` will handle a `ChildFinished` or `ChildPanicked` message and can restart or cleanup the child.

- **Tokio (Std) Patterns:** In a Tokio environment, we can leverage more dynamic tools. Each actor runs as an async task on the Tokio executor, so a supervisor can hold `JoinHandle`**s or use a** `JoinSet` to await child task termination. A common pattern is linking actors: when spawning a child, store its `JoinHandle` and also send the child a clone of the parent's `Address` for error reporting. If the child's future completes normally, it can send a "stopped" message to the parent; if it panics, Tokio (with unwinding) will just drop the task. Tokio doesn't automatically restart tasks, but one can catch panics by spawning the actor within a `catch_unwind` wrapper (requires `UnwindSafe`). Frameworks like **Actix and Ractor implement linking**: Ractor's supervisor gets notified on child start/stop/fail if panics are caught (unless `panic = abort`) [1]. Actix provides a `Supervisor` struct that automatically restarts a failed actor by calling its `restarting()` hook [4] (the failed message is lost but the actor is alive again). Bastion goes further with a hierarchy of supervisors and a default *one-for-one* strategy that **automatically supervises tasks** on spawn [5].

- **Design in `lit-bit`:** The `lit-bit` actor model aims for an OTP-style supervision tree on both platforms. Each actor can declare a `RestartStrategy` (`OneForOne`, `OneForAll`, `RestForOne`) determining how a supervisor handles failures [6]. Internally, the `Address<T>` **type holds parent/child links** (in the Tokio variant, an actor's address keeps a `Weak` reference to its parent's control block and a list of child `Weak` references [7]). This means when an actor is spawned under a supervisor, it registers itself, enabling the parent to track and restart children. The *supervisor lifecycle API* includes `on_start`, `on_stop`, and `on_panic` hooks on the Actor trait [8]. For instance, if an actor's `on_panic` returns `RestartStrategy::OneForOne`, the runtime should notify the parent and restart that actor alone [9] [10]. The implementation might wrap each actor task in a catch mechanism (in std) that on exit sends a `SupervisorMessage::ChildPanicked{id}` to the parent's mailbox. In no-std, true panic catching isn't feasible, so we rely on actors to *fail gracefully*: e.g. a sensor actor encountering a critical error can send `ChildPanicked{id}` to its supervisor (instead of panicking) and then end its loop. The supervisor's `on_event` would handle this by restarting the child (spawning a new actor instance) [11]. This approach ensures **deterministic message flow** – the order of failure notifications relative to other messages is well-defined, and no extra threads or dynamic selects are used. It mirrors Erlang's "let it crash" philosophy but adapted to Rust: on embedded, "crashing" means sending an error up rather than aborting.

- **Comparison Summary:** In Actix's model, a supervisor sits alongside the actor and calls a restart hook on failure (the failed message is dropped with an error to the sender) [4]. Ractor's actors are linked so that the supervisor's `handle_supervisor_evt` is invoked on child termination (except in abort scenarios) [12]. Bastion provides a full tree: supervisors manage children groups and apply a policy (e.g. restart up to N times, or escalate). `lit-bit` will follow the same principles but under severe constraints: *no allocation*, *no unwinding on embedded*, and *bounded memory*. This means careful design of the **supervisor message protocol** (using, for example, a simple enum like `SupervisorMessage::ChildPanicked{id}`) and using fixed-size data structures (e.g. an array or `Vec<Address<..>>` of children pre-allocated to some max). Embassy's static memory model encourages defining the maximum number of children at compile time or using a `heapless::Vec`. Ultimately, async supervision is realized by **hierarchical actor relationships and explicit failure messages**, rather than polling multiple futures. This achieves robust supervision without `select_all`, aligning with the OTP-inspired restart strategies in `lit-bit` [13].

2. Zero-Allocation Message Batching

Goal: Increase throughput by processing messages in batches from each actor's mailbox, without allocating and without breaking the semantics of single-message handling. In `lit-bit`, mailboxes are bounded SPSC queues (fixed capacity) – using `heapless` for no-std and Tokio's MPSC for std [14]. Normally, an actor processes one message at a time (ensuring each event is handled to completion in order [15]). Batching means taking multiple queued messages in one go and handling them in sequence before yielding.

- **Using `heapless::spsc::Queue::dequeue_many`:** The heapless SPSC queue offers methods to efficiently dequeue multiple items. Although not explicitly named in docs, the internal design allows borrowing a **contiguous slice** of available messages. This is analogous to patterns in high-performance messaging systems: for example, the CoralQueue in Java requires checking how many items are available, then fetching a batch in one operation – *"This design allows the ring to perform*

batching naturally. And batching is very important for performance." [16]. By dequeuing many messages at once, we minimize per-message overhead (locking, checking pointers, etc.) and exploit CPU cache locality. In practice, an actor's event loop can be written to drain the queue until empty: e.g.

```
async fn on_event(&mut self, _: ()) {
    // This might never actually be called; our actor uses on_batch instead.
}
// Pseudo-code for internal event loop:
loop {
    // Wait for at least one message (async block on recv or semaphore)
    inbox.next().await;
    // Then dequeue as many messages as available without awaiting again:
    while let Some(msg) = inbox.try_dequeue() {
        self.handle_msg(msg);
        batch_count += 1;
        if batch_count >= MAX_BATCH {
            break; // optional: limit batch size to yield periodically
        }
    }
    // After draining or hitting batch limit, yield back to executor here.
}
```

In embedded, one could use `queue.dequeue_many()` to get a slice of messages currently in the buffer and process them in a tight loop. This **zero-copy batch** read is possible because the SPSC queue stores messages in a circular buffer. By processing contiguous messages as a batch, we amortize the cost of waking the task. This approach is similar to how **ZeroMQ** achieves high throughput: *"if you feed it messages very rapidly, it sends the first immediately and queues the rest in user-space... then it sends all the queued messages as a batch. So you get both high throughput and low latency."* [17] In our context, once the actor wakes up, it will handle not just one message but possibly dozens, then yield, improving throughput.

- **Optional Batching Semantics:** We must preserve the logical behavior as if messages were one-by-one. Batching should be an *optimization*, invisible to the sender. In-order delivery is still guaranteed – we simply take multiple messages in FIFO order and process them consecutively. No message is dropped or re-ordered. The difference is that an actor might not suspend after a single message if more work is already queued. This is safe as long as each message's `on_event` is independent and the actor doesn't need to receive interleaving from other tasks in between. To keep fairness, a **batch size limit or timer** can be used: e.g. process at most N messages or for at most M milliseconds in one go, then yield. This prevents one actor with a hot inbox from starving others on the same thread (especially relevant in Embassy's single-thread executor). Tokio's cooperative scheduling already has a budget mechanism to prevent long-running tasks from blocking the scheduler – our batching should respect that by periodically `.await`ing (which yields). On Embassy, we manually ensure no infinite loop without yield.

Importantly, single-message semantics like back-pressure remain intact. For example, if an actor's mailbox is near full, senders will experience back-pressure (Tokio's `send().await` will wait, Embassy's `try_send` returns `Full`) [18] [19]. Batching doesn't change this – it only means the consumer (actor) pulls more data

per wake-up. We also maintain determinism: a batch of 5 messages will have the same effect as 5 individual `await` calls in sequence, except with lower overhead.

- **Throughput Benefits and Measurement:** Batching reduces context-switching and per-message call overhead. In a microcontroller at 120 MHz, processing one message at a time might handle e.g. 50k messages/sec. By batch-processing, we might significantly increase this. Prior research in `lit-bit` set performance targets of **>100k msg/sec per core** and under **200 ns latency per message** on desktop [20] [13]. Batching helps achieve these numbers. To verify improvements, we can benchmark in both environments:
- *Desktop/Tokio:* use a high-frequency message sender (or a loopback actor) to send, say, 1e6 messages and measure the time with batching on vs off. Tokio's multi-threading might even allow >1 million msgs/sec with batching. We can use `tokio::time::Instant` for timing or an external criterion benchmark.

- *Embedded/Embassy:* instrumentation is trickier without std I/O. We can toggle a GPIO pin each time a batch is processed (e.g. pulse high after every 100 messages) and measure the frequency with a logic analyzer. Alternatively, use a hardware timer or cycle counter (ARM's DWT cycle counter) to measure cycles per message. The expectation is that batching yields a higher throughput. For example, if single-message handling costs 10 µs each, processing 5 at once might only cost 50 µs total instead of 5× context switches plus overhead. We might observe, say, a 20-30% throughput increase by reducing scheduler wake-ups. Real-world improvement depends on message size and actor workload; if the actor's `on_event` is very light, batching provides a bigger relative win (since message overhead dominates). If `on_event` is heavy, the benefit is smaller but still present (fewer awaits).

- **Comparisons:** Other systems employ batching for performance:

- **ZeroMQ** (as noted) batches messages in its internal queues to maximize socket throughput [17].
- **Chronicle Queue** (Java) is effectively a file-backed ring buffer; it can sustain *hundreds of thousands of messages/sec* by writing/reading in large chunks contiguously [21]. It shows that a sequential, batched approach to queue IO minimizes latency spikes. A Chronicle dev note shows *"without batching, Chronicle takes over twice as long"* for certain operations, underlining batching's impact [22].
- **Riker and Actix:** These Rust actor frameworks did not explicitly advertise message batching in their design. Actix processes one message at a time per actor (with an optional mailbox capacity limit for back-pressure). Throughput in Actix can be high, but it relies on fast message passing under the hood. There's anecdotal evidence that high-performance Rust actors (like the **Steady** framework) incorporate batching; Steady State's documentation explicitly lists *"support for batch processing in actors"* as a feature for high throughput [23]. This trend shows that to reach millions of messages/sec, batching is key.
- **Embedded frameworks:** Ector (an Embassy-based actor framework) doesn't mention batching – it awaits each message one by one [24]. Our approach in `lit-bit` would go beyond by optionally draining the inbox. In constrained devices, this can significantly boost efficiency when many events are queued (common in sensor hubs or telemetry aggregators).

Implementing optional vs default: We can provide batching as an **opt-in feature**. By default, `Actor::on_event` processes one message. But we could add an `Actor::on_batch(&mut self, msgs: &mut [Self::Message])` for batch handling. If an actor implements `on_batch`, the runtime could fill an array or slice of messages (using `dequeue_many`) and call `on_batch` instead of many

individual `on_event` calls. This maintains backwards compatibility – old actors ignore the feature. Another lighter approach is simply internally looping on the queue as shown, without changing the trait: this is transparent to the actor code but yields the benefits of batching. The *lit-bit* docs already encourage a form of batching manually (collecting messages in a buffer until `BATCH_SIZE` then processing) [25]; the new mechanism would automate that via the mailbox. In sum, message batching can be achieved with zero allocations (reusing the fixed buffer), and it should be configurable per actor use-case to avoid surprising behavior.

3. Cross-Runtime Trade-offs: Tokio vs. Embassy

Supporting both Tokio (for desktop/Server) and Embassy (for embedded) with one codebase introduces trade-offs in scheduling, memory, and performance:

- **Scheduling and Concurrency:** Tokio is a multithreaded, preemptive executor (if configured as such). It can run many actor tasks in parallel on different CPU cores. This yields extremely high throughput in aggregate and true concurrent execution of actors. Embassy, by contrast, runs on a single core (most microcontrollers) with a **cooperative scheduler** – tasks run to completion or until they `.await`. Thus, **Tokio can exploit parallelism**, while Embassy ensures **deterministic interleaving** (no two actors run at the exact same time). For `lit-bit`, the actor model is single-threaded *per actor* [15], but globally Tokio can achieve parallel message processing across actors, whereas Embassy will strictly serialize across ready tasks. The implication: a busy actor on Embassy can delay others if it doesn't yield (hence the importance of not hogging the executor – use short batches and frequent awaits). On Tokio, the OS scheduler and Tokio's work-stealing will balance actors across threads, so one actor's heavy load won't necessarily block another (unless all are on one thread).

- **Memory Overhead:** Tokio tasks and channels use heap allocations. For example, a Tokio actor with a mailbox and state was measured around **2 KB** memory usage (task stack, heap allocations for channel, etc.) [26]. Embassy tasks are zero-alloc: the stack for each task is a fixed compile-time stack (often placed in `.bss` or static memory), and the `embassy::channel` uses a fixed-size buffer. A minimal no_std actor in `lit-bit` was about **64 bytes** (task overhead) and even a state machine actor ~128 bytes [27] – **orders of magnitude smaller** than a Tokio actor. This is crucial for embedded targets where RAM is measured in KB. However, the flip side is inflexibility: in Embassy you must decide buffer sizes and stack sizes ahead of time. Tokio can grow and shrink with load (but consumes more baseline memory). Maintaining a *platform-dual* code without divergence means we design for the lowest common denominator: no dynamic memory, and explicit sizing. Thus, even on Tokio, `lit-bit` might avoid unbounded channels or large heap use, sticking to the same fixed-size mailbox API (Tokio's `mpsc` is used but our API can enforce capacity). This unified approach ensures consistent semantics. Conditional compilation is used internally (different `Address` impl for each backend [14]), but from the user perspective the API is the same.

- **Throughput and Latency:** On desktop, Tokio can push extremely high message rates. A single actor on one core can handle ~1 million messages/sec (as per design targets) [13], and multiple actors on multiple cores scale that out. Embassy, limited by CPU speed (e.g. 200 MHz ARM) and single-core, might handle on the order of 100k messages/sec in a tight loop [20]. The latency per message might actually be comparable in absolute terms: <200 ns on a 3.5 GHz CPU vs perhaps a few microseconds on a 200 MHz MCU (since clock is slower). The **relative overhead** of the runtime is small in both

cases because of zero-copy messaging. One difference is that **Tokio introduces some latency jitter** due to thread scheduling and contention, whereas Embassy's strictly deterministic loop means if an actor yields, the next ready message is processed next with minimal overhead. In practice, Tokio's latency can still be very low, but for real-time needs, Embassy gives more predictability (no contention, no OS preemption – just interrupt latency to consider). `lit-bit` aims to keep message handling **consistent**: whether on Tokio or Embassy, one message = one iteration of the actor's event loop. Batching (if enabled) benefits both, though on a multi-core system the benefit might be slightly less (since even without batching, multiple messages could be processed concurrently by splitting actors onto threads).

- **Back-Pressure and Deadlock:** A subtle trade-off is that **Embassy channels never signal closure** [28]. If an actor is dropped on Embassy, its mailbox sender might block forever rather than get a closed error. On Tokio, if an actor (receiver) is dropped, senders get a `Closed` error. This means supervision must handle mailbox teardown differently. On Embassy, dropping a child actor's task *without rebooting* is unusual – typically all tasks run until a reset. If we do want to stop an actor (e.g. a child that we will restart), we might need to consume or discard any pending messages manually because the channel won't close itself. Tokio's runtime makes it easier to drop an actor and its mailbox (the `mpsc` will close and any pending messages can be dropped or drained). We maintain platform-dual code by abstracting these differences: for instance, providing an `Address::stop()` that on Tokio drops the receiver and on Embassy perhaps triggers a flag for the task to end. Ensuring no memory leaks on Embassy (since tasks are static, they might not free resources until reset) is also critical – ideally each actor holds no heap memory so "leaking" an actor just means it stops processing, which in an MCU scenario might be acceptable if infrequent.

- **Unified Abstractions vs Conditional Code:** The design strives for *no forks in the public API*. The same `Actor` trait and `Address` type work in both modes. Under the hood, we have conditional implementations (as seen with `#[cfg(async-embassy)]` vs `#[cfg(async-tokio)]` in the address and spawn code). This avoids a complex trait-object abstraction for mailboxes – at compile time we pick the implementation [29]. The developer using `lit-bit` mostly doesn't need to worry about these differences beyond enabling the correct feature flag. Writing the actor logic is identical, and you can target `thumbv7m-none-eabi` or `x86_64-unknown-linux-gnu` just by toggling features. We do, however, recommend testing on each target because performance characteristics differ. For example, an actor that is optimal on Embassy (small, batch-processing to reduce wakeups) will also work on Tokio, but one might find that Tokio can handle a larger mailbox without problems whereas on embedded a huge mailbox is a "memory bomb" [30]. Thus we document best practices (e.g. don't use mailbox size 100k on embedded – use 32 or so for natural back-pressure [30]). The table below summarizes some key differences and strategies:

Aspect	**Tokio (std)**	**Embassy (no_std)**
Task spawn & run	Dynamic tasks on heap; `JoinHandle` for each actor. Can use many OS threads (parallel execution).	Static tasks (via `Spawner`) with fixed stack in memory. Single thread (cooperative multitasking).

Aspect	**Tokio (std)**	**Embassy (no_std)**
Panic behavior	Panic in actor task logs error; task stops. Can catch with `Supervisor` (actor restart) [4] or by `join.await` if using `Result`. Unwinding is possible (if enabled).	Panic typically aborts the system (no unwind). Should be treated as fatal – prefer error handling within actor. Use `on_panic` hook to decide restart policy, but actual catch is via supervisor message (if any).
Task cancellation	Supported via `JoinHandle::abort()` (forces drop of the future). Actor can also observe `Context::stop()` in Actix, etc. Cancellation is cooperative if using your own flag.	No built-in cancel. Must signal the task to stop (e.g. set a global flag or send a special message it listens for). Dropping tasks isn't automatic; you design a "stop condition" in the actor's loop. Embassy's executor lacks a direct kill switch (community request noted) – cancellation is by contract within the code.
Mailbox	`tokio::sync::mpsc` channel. Heap allocated ring buffer. Sends can fail if receiver is dropped (you get `SendError` on closed). Capacity can be unbounded or bounded (we use bounded for back-pressure).	`embassy_sync::channel::Channel` with static buffer of size N. Never signals closed (no `Closed` error) [3]. If receiver task ends, sender will just block forever on a full buffer. Therefore, design assumes actors live for system lifetime or supervisor explicitly handles removal.
Memory overhead	Higher: ~2 KB per actor for stack + heap structures (example) [26]. Grows with more actors, messages allocate within channel (but message data itself is on heap or stack depending on usage).	Minimal: tens of bytes per actor (task state + static queue) [27] plus the fixed mailbox array. Memory cost is mostly the mailbox buffer (e.g. 32 messages * size of message). No dynamic growth – memory usage is predictable.
Throughput	Extremely high on capable hardware. Multiple actors can run truly concurrently. With batching and proper tuning, millions of msgs/sec on modern CPUs are achievable [13]. Latency ~sub-microsecond per message in optimal cases [20].	High for an MCU but limited by clock speed and single-core. Tens or hundreds of thousands msgs/sec in tight loop (e.g. 100k/sec observed target) [20]. Batching helps approach that. Interrupts and hardware operations can introduce latency outliers, but within the actor system, timing is very consistent.

Aspect	**Tokio (std)**	**Embassy (no_std)**
Maintaining one codebase	Use feature flags to switch implementations. Test both. Minor conditional logic in code (e.g. `cfg(if_embassy) { spawn_embassy(...) } else { spawn_tokio(...) }`). Avoid std-only calls in core logic (use `alloc` only behind feature if needed). The actor API abstracts differences so business logic is identical.	Same code as Tokio except at spawn and in HAL interactions. Ensure no std assumptions (e.g. use `Instant` from embassy time instead of std). Keep allocations out (no `Vec` or `Box` unless behind an off-by-default feature). The code structure (spawn, address, message handling) remains unified – no need to fork the project for embedded.

By balancing these differences, `lit-bit` can truly be *platform-dual* – the **same actor code runs on an Arm Cortex-M bare-metal or on Linux** [31]. Developers get confidence that the supervision and messaging semantics are consistent across environments, with only performance tuning differing.

4. Code Examples and Integration

Below are focused examples illustrating supervision structures, batching APIs, and using `Address<T>` in both Embassy and Tokio contexts. These are simplified for clarity:

Example 1: Supervisor with Child Actors (Embassy style)
This example shows a supervisor actor that spawns child actors and monitors them. It uses a simple message to indicate a child panic/failure, and restarts the child. (In practice the framework might automate the notification, but here we do it manually for illustration.)

```
# use lit_bit_core::actor::{Actor, Address, RestartStrategy};
# // Assume appropriate Embassy executor context
enum SupervisorMessage {
    ChildPanicked { id: u32 },
    StartChild(u32)
}

struct MySupervisor {
    children: heapless::Vec<(u32, Address<ChildMsg, 8>), 4>,  // up to 4
children
}
impl Actor for MySupervisor {
    type Message = SupervisorMessage;
    async fn on_event(&mut self, msg: SupervisorMessage) {
        match msg {
            SupervisorMessage::StartChild(child_id) => {
                // Spawn a new child actor under Embassy
                let child = ChildActor::new(child_id);
```

```
                let addr = spawn_actor_embassy!
(child);  // pseudo-code macro to spawn
                self.children.push((child_id, addr)).ok();
            }
            SupervisorMessage::ChildPanicked { id } => {
                // Find and remove the old child (if still in list)
                if let Some(index) = self.children.iter().position(|(cid, _)|
*cid == id) {
                    self.children.swap_remove(index);
                }
                // Restart: spawn a new ChildActor with the same id
                let new_child = ChildActor::new(id);
                let new_addr = spawn_actor_embassy!(new_child);
                self.children.push((id, new_addr)).ok();
                // (Optionally, log or send an event that restart happened)
            }
        }
    }
}

// Child actor that deliberately notifies supervisor on error
enum ChildMsg { DoWork(u32), Fail }
struct ChildActor { id: u32, /* ... */ }
impl ChildActor { fn new(id: u32) -> Self { Self { id } } }
impl Actor for ChildActor {
    type Message = ChildMsg;
    async fn on_event(&mut self, msg: ChildMsg) {
        match msg {
            ChildMsg::DoWork(data) => {
                // ... normal work ...
            }
            ChildMsg::Fail => {
                // Simulate a failure condition:
                // Notify supervisor of "panic" and stop
                let sup_addr = /* obtain supervisor Address somehow */;
                let _ = sup_addr.try_send(SupervisorMessage::ChildPanicked {
id: self.id });
                // The actor will terminate after returning (no further .await,
drop self)
            }
        }
    }
    fn on_panic(&self, _info: &core::panic::PanicInfo) -> RestartStrategy {
        // Even if a real panic occurs, tell runtime to restart this actor alone
        RestartStrategy::OneForOne
    }
}
```

In this snippet, `spawn_actor_embassy!` is assumed to spawn the actor on the Embassy executor (and likely takes a `Spawner`). The supervisor manages a static list of children addresses. When a child wants to simulate a failure, it sends a `ChildPanicked` message to its parent. The supervisor then restarts the child. In a full system, the runtime would wire this up so that an actual panic in `ChildActor` triggers the same message (perhaps via `on_panic` hook). Note how we avoid dynamic allocation: the children list is a fixed-capacity `heapless::Vec`, and each `Address` uses a fixed mailbox of 8 messages. This ensures zero runtime allocation.

Example 2: Batching Messages in an Actor
Below, `BatchProcessor` actor processes incoming work items in batches. We demonstrate two approaches: an explicit batch in user code, and a conceptual use of `dequeue_many`.

```
struct BatchProcessor {
    batch: heapless::Vec<WorkItem, 16>,  // Buffer to collect a batch of
messages
}
impl Actor for BatchProcessor {
    type Message = WorkItem;
    async fn on_event(&mut self, item: WorkItem) {
        // Collect the item into a batch
        self.batch.push(item).ok();
        if self.batch.len() >= BATCH_SIZE {
            // Process the batch when full
            self.process_batch().await;
            self.batch.clear();
        }
        // If batch not full, wait for more messages
        // (The actor will automatically await the next message)
    }
}
impl BatchProcessor {
    async fn process_batch(&mut self) {
        // Example batch processing logic
        for work in self.batch.iter() {
            // ... handle each work item ...
        }
        // e.g., send results or commit all at once
    }
}
# struct WorkItem;
```

This code follows the pattern in the docs [25]: accumulate messages and then process in one go. It ensures fewer async yields (one `await` per batch rather than per message). It's zero-allocation because `heapless::Vec` is used for the batch buffer. The downside is manual batching logic in every actor that needs it.

Using the mailbox's batch API, we could simplify this. Suppose `Address`/inbox had a method to fetch many messages at once – the actor's internal loop (managed by the runtime) might do:

```
// Pseudo-code for runtime's polling of mailbox
if inbox.ready() {
    // Grab multiple messages in one slice
    inbox.dequeue_many(|msgs: &mut [WorkItem]| {
        // Here we got a slice of pending WorkItem messages
        batch_processor.process_batch_slice(msgs);
        // The closure can return number of msgs processed, which dequeue_many
will pop
        msgs.len()
    });
}
```

And `process_batch_slice` could simply iterate the provided slice (already a batch of messages). This avoids even pushing to a second buffer – we process straight from the queue's memory. The closure returns `msgs.len()` to tell the queue that all those messages were consumed. This pattern is akin to how lock-free ring buffers allow batch processing for efficiency [16]. It maintains single-threaded access (the closure executes within the queue's critical section or with interrupts disabled, etc., to safely read multiple elements).

Example 3: Cross-Runtime Address Usage
Finally, sending messages via `Address<T>` looks the same on both runtimes, with subtle differences in behavior:

```
# use lit_bit_core::actor::SendError;
# #[derive(Debug)] struct SensorReading(i32);
# let tokio_addr: Address<SensorReading, 32> = /* ... */ unimplemented!();
# let embassy_addr: Address<SensorReading, 32> = /* ... */ unimplemented!();
// Tokio (std) usage:
if let Err(SendError::Closed(msg)) = tokio_addr.send(SensorReading(42)).await {
    eprintln!("Actor is probably stopped, message {:?} lost", msg);
}
// Embassy (no_std) usage:
match embassy_addr.try_send(SensorReading(42)) {
    Ok(()) => { /* message queued */ }
    Err(SendError::Full(msg)) => {
        // Mailbox full – actor is slow or dead. In Embassy, we can't
distinguish.
        // Handle by dropping or retrying later.
        log_warn!("Mailbox full, dropping {:?}", msg);
    }
```

```
    Err(SendError::Closed(_)) => unreachable!("Embassy channels never close")
}
```

As shown, on Tokio `send().await` will wait for space and can error only if the receiver is gone [32]. On Embassy, `send().await` never errors (it will wait indefinitely if the queue is full) [33], so a non-blocking `try_send` is used to avoid deadlock. The `SendError::Closed` variant is essentially unused in no_std builds [19]. This example highlights how the same API calls are utilized under the hood, while the implementation ensures the right behavior per platform.

Minimal Dependencies and Integration: All the above patterns avoid adding any new crates beyond what we already have (heapless, embassy, tokio). We reuse existing concepts: Embassy's channel provides the fixed-size queue, and we leverage Rust's async/await semantics for scheduling. No heap means we must carefully pick data structures (e.g. use `heapless::Queue` or `Vec` for buffers, avoid `Vec::with_capacity` at runtime). By adapting patterns from frameworks (Actix's supervisor, Bastion's restart strategies, ZeroMQ's batching) and using them in a `no_std` style, we create an actor system that is both **efficient and resilient**. The end result is that `lit-bit` actors can achieve high performance – on par with or exceeding traditional frameworks – while running on anything from an ARM Cortex-M microcontroller to an x86_64 server, **with the same code** [31].

Sources: The design principles and examples above draw on the `lit-bit` project documentation and similar actor systems. For instance, `lit-bit`'s roadmap emphasizes a supervision tree and cross-platform mailbox design [13], and its guides illustrate batching for throughput [25] and avoiding spawn-without-supervision pitfalls [34]. We also referenced how other frameworks handle these problems: Actix's Supervisor for actor restart [4], Ractor's linked supervision model [12], and Bastion's one-for-one strategy [5]. The importance of batching is well-known in messaging systems (e.g. ZeroMQ's batching of messages to boost throughput [17] and lock-free queue research [16]). These informed our approach to implement message batching in a zero-allocation context. All these techniques coalesce into the `lit-bit` actor system, fulfilling its promise of **"mature actor model patterns"** with **no allocation, deterministic supervision, and high performance** on both embedded and async runtimes [35] [20].

---

[1] [12] ractor - Rust
https://docs.rs/ractor/latest/ractor/

[2] [3] [7] [18] [19] [28] [32] [33] address.rs
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/lit-bit-core/src/actor/address.rs

[4] Supervisor in actix - Rust
https://docs.rs/actix/latest/actix/struct.Supervisor.html

[5] GitHub - bastion-rs/bastion: Highly-available Distributed Fault-tolerant Runtime
https://github.com/bastion-rs/bastion

[6] [9] [10] [11] [20] [25] [26] [27] [30] [31] [34] [35] actor-overview.md
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/docs/actor-overview.md

[8] [13] [14] [15] [29] Spec.md
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/Spec.md

16 Ultra-low-latency, batching and concurrent queue for IPC in Java | Hacker News
https://news.ycombinator.com/item?id=42213845

17 I've played with ZeroMQ on some small projects, and I've been quite impressed by... | Hacker News
https://news.ycombinator.com/item?id=1660695

21 Big Data and Chronicle Queue
https://chronicle.software/big-data-and-chronicle-queue/

22 Java Chronicle Performance - Google Groups
https://groups.google.com/g/java-chronicle/c/lYyJZAu1TXs

23 SteadyState — Rust network library // Lib.rs
https://lib.rs/crates/steady_state

24 ector - Rust
https://docs.rs/ector/latest/ector/