ChatGPT

Safe Patterns for Static Mailboxes in `no_std` Rust

In **embedded Rust** (no-std), designing a mailbox (message queue) for actor systems requires careful handling of memory. We want a **statically allocated** SPSC (single-producer, single-consumer) queue that doesn't use the heap, yet is safe and ergonomic for users. Below, we explore how existing libraries tackle this, and outline patterns (including macros, builders, and newtypes) that achieve **safe static mailboxes** without forcing the user to write `unsafe` or leak memory. We focus on SPSC queues (one producer, one consumer), which simplifies design and avoids unnecessary overhead.

Existing Approaches to Static SPSC Queues in Embedded Rust

Several established embedded Rust libraries already support static, no-heap queues:

- **Heapless `spsc::Queue`:** The `heapless` crate provides a lock-free ring buffer for SPSC communication. You can declare a `Queue<T, N>` in a `static` context because it stores its buffer inline (capacity `N` is a const generic) [1] [2]. For example, one could do:

```
static mut Q: heapless::spsc::Queue<Event, 4> = heapless::spsc::Queue::new();
```

This creates a fixed-capacity queue of `Event` that lives in static memory. To use it as an SPSC channel, you **split** the queue into a `Producer` and `Consumer`. The split operation consumes a mutable reference to the queue and returns two endpoint handles. However, obtaining a `&'static mut Queue` from a `static mut` requires unsafe code. The Heapless docs demonstrate this pattern, where the user takes an `unsafe { &mut Q }` and then calls `split()` [3] [4]. This yields `Producer<'static, T, N>` and `Consumer<'static, T, N>` which can be sent to different contexts (e.g. an ISR vs. a main loop) for lock-free enqueue/dequeue [4] [5]. The downside is the user must write `unsafe` to get the static reference (ensuring it's only taken once).

- **RTIC framework:** The *Real-Time Interrupt-driven Concurrency* (RTIC) framework uses macros to safely allocate and share static data between tasks. In RTIC 1.x and 2.x, you can declare a static `heapless::spsc::Queue` in an `#[init]` context that automatically gives a `'static` lifetime. For example, RTIC 1.x allowed:

```
#[init(local = [q: Queue<u32, 5> = Queue::new()])]
fn init(cx: init::Context) -> ... {
    // `q` is allocated statically and lives beyond `init`
    let (p, c) = cx.local.q.split();
    ...
}
```

Here, the RTIC macro ensures `cx.local.q` has `'static` lifetime after `init` returns [6] [7]. The queue is defined as a *local resource* in `init`, so it's placed in static memory, and splitting it yields a `Producer<'static, ...>` and `Consumer<'static, ...>` that can be moved to different tasks safely [6] [8]. **No user `unsafe` is needed** – the macro expansion handles the static and the lifetime guarantees. RTIC's approach is very ergonomic: tasks can list the producer/consumer in their resource lists and use them directly, with RTIC enforcing at compile-time that only the producer task calls `enqueue` and only the consumer task calls `dequeue`. This pattern is used, for instance, to share an event queue between an interrupt handler and a background task without locks.

- **Embassy and Ector (async actor frameworks):** The `embassy` async framework and its actor library `ector` also avoid heap allocation and use static memory for channels. **Ector** (an Embassy-integrated actor framework) provides an `Inbox<M>` trait and related types for message passing, built on static capacity channels. Under the hood, Ector relies on a statically allocated channel (similar to a ring buffer) for each actor's inbox. In practice, this is achieved via something like `StaticChannel` from the `thingbuf` crate. The `thingbuf` crate offers a **statically-allocated MPSC channel** and a lock-free queue for no_std use [9] [10]. For example, using thingbuf's asynchronous StaticChannel:

```
// In a no_std context (no allocator available)
static KERNEL_EVENTS: thingbuf::mpsc::StaticChannel<KernelEvent, 256>
    = thingbuf::mpsc::StaticChannel::new();

// ... later, split into sender and receiver
let (event_tx, event_rx) = KERNEL_EVENTS.split();
```

This is a **const-initialized channel** of 256 `KernelEvent` slots, living in `.bss` or `.data`. Calling `split()` returns a `StaticSender` and `StaticReceiver` with `'static` lifetimes [11] [12]. The `StaticChannel` internally manages the ring buffer and ensures `split()` is only called once (subsequent calls would typically panic or return an error). Embassy's actors (Ector) likely use a similar construct for each actor's inbox, giving each actor its own static SPSC queue for incoming messages. The user doesn't interact with unsafe or with the queue directly – they use an actor's address or sender handle to send messages. **Key point:** crates like *thingbuf* encapsulate the static queue pattern and expose a safe API (similar to heapless, but without requiring the user to do the `unsafe` dance).

- **BBQueue:** Another crate, `BBQueue`, provides a statically allocated SPSC queue designed for DMA transfers (using a bipartite buffer). It also supports const initialization. For example, one can do `static BB: BBBuffer<6> = BBBuffer::new();` and later `let (prod, cons) = BB.try_split().unwrap();` [13] [14]. BBQueue's `try_split()` returns an error if called more than once [15], which is a runtime check to prevent aliasing the buffer. This is another example of enforcing safety: the queue can only be split into one producer and one consumer, preserving the SPSC model. BBQueue is especially useful when you need to get contiguous blocks for DMA, but as a pattern it shows **const-initialized, static queue with a safe split**.

- **User-space patterns (manual static mut):** Without using a framework or special crate, many users do manually what Heapless suggests: define a `static mut Queue` and obtain a mutable static reference once. For instance, the *Drogue IoT* project's blog shows two static queues defined at the top-level and passed into an initialization function as `unsafe { &mut RESPONSE_QUEUE }` [16]. This works, but it's not idiomatic because it forces user-level unsafe code. The goal in modern embedded Rust is to avoid requiring the end user to write `unsafe` to set up their mailboxes.

Summary: Established solutions either use **macros or dedicated types** to hide the `unsafe`. RTIC macros and thingbuf's `StaticChannel` ensure the queue is allocated statically and only split once. Heapless gives the raw tools (const constructors and split) but leaves safety to the user or higher-level frameworks. Next, we'll delve into macro patterns and how to wrap these queues more ergonomically.

Macro-Based Wrappers for Static Mailboxes

Macros can greatly improve ergonomics and safety by handling static declarations and splitting internally. A macro can allocate the queue in a static scope and perform the `split()`, returning the producer/consumer pair to the user. This means the user doesn't write `unsafe` or worry about lifetime annotations.

Example – RTIC's `make_channel!`: The RTIC project provides the `rtic_sync::make_channel!` macro to create a channel with `'static` lifetime [17] [18]. This macro is geared towards an **MPSC** channel (multiple producers, single consumer) for async tasks, but the concept is similar for SPSC. Internally, `make_channel!` expands to something like:

```
macro_rules! make_channel {
    ($type:ty, $size:expr) => {{
        static mut CHANNEL: rtic_sync::channel::Channel<$type, $size>
            = rtic_sync::channel::Channel::new();
        static CHECK: AtomicU8 = AtomicU8::new(0);
        critical_section::with(|_| {
            if CHECK.load(Ordering::Relaxed) != 0 {
                panic!("call to the same `make_channel` instance twice");
            }
            CHECK.store(1, Ordering::Relaxed);
        });
        // SAFETY: Only one mutable access; we hide `CHANNEL` from the user.
        unsafe { CHANNEL.split() }
    }};
}
```

When you invoke `make_channel!(T, N)`, it returns `(Sender<'static, T, N>, Receiver<'static, T, N>)` by defining a hidden static queue and splitting it [19] [20]. The macro even uses a `CHECK` flag to panic if it's invoked more than once for the same static (preventing a double split which would be unsound). This pattern shows how a macro can wrap the static definition and `unsafe` in a safe interface. The user simply writes `let (tx, rx) = make_channel!(MyMsg, 8);` inside their init function, and they get back channel endpoints with no unsafe code needed. (In RTIC's case these are async-capable Sender/Receiver types, but one could analogously create a macro for a simple SPSC heapless queue.)

Custom macro for Heapless SPSC: It's feasible to write a small macro for `heapless::spsc::Queue` specifically. For example, a macro could expand to a static `Queue` and return the producer/consumer:

```
/// Defines a static SPSC queue and returns producer & consumer.
#[macro_export]
macro_rules! static_mailbox {
    ($name:ident: $t:ty, $capacity:expr) => {{
        static mut $name: heapless::spsc::Queue<$t, $capacity>
            = heapless::spsc::Queue::new();
        // Allow a mutable static reference to be taken once.
        #[allow(unused_unsafe)]
        let queue_ref: &'static mut heapless::spsc::Queue<$t, $capacity>
            = unsafe { &mut $name };
        queue_ref.split()
    }}
}
```

Using this hypothetical macro, a user could do `let (prod, cons) = static_mailbox!(MYQ: Event, 16);` in a setup function. The macro hides the `static mut` and the `unsafe` block required to get `&'static mut`. You'd likely include similar one-time initialization checks as RTIC does (to avoid splitting twice). This macro-based approach ensures *at compile time* that the queue lives forever (because it's static) and that the user can't easily misuse the raw queue (they never get `&mut` access to it, only the safe producer/consumer).

Attaching memory attributes via macros: One challenge with macros is allowing the user to control memory placement (e.g., placing the static in a particular linker section). A well-designed macro can accept attributes. For instance, we could allow `static_mailbox!` to take an optional attribute:

```
static_mailbox!(
    #[link_section = ".sram3"] // place in a specific memory region
    MYQ: Event, 16
);
```

The macro would then apply `#[link_section = ".sram3"]` to the static definition it generates. This way, the user retains control over where the queue is located (important for embedded scenarios where certain memory is faster or has special properties).

Takeaway: Macros can make static mailbox setup **declarative and safe**. They improve ergonomics by reducing boilerplate and preventing common mistakes (like forgetting `unsafe` or splitting a queue multiple times). Many frameworks (like RTIC) use macros internally for exactly this reason. If designing an actor system, providing a macro to create a mailbox can simplify the API. Just be mindful to document what the macro does and consider how users might specify memory attributes or capacities in a flexible way.

Idiomatic Patterns: Builders and Abstractions Without Macros

Beyond macros, we can achieve safe static allocation using patterns like **builders, newtype wrappers, or dedicated structs** that encapsulate unsafe code. The goal is to internalize any `unsafe` within a well-audited abstraction and present a safe interface.

- `StaticCell` **and one-time initialization:** The `static_cell` crate provides a small type `StaticCell<T>` for exactly this use case. It lets you allocate memory at compile-time and initialize at runtime, returning a `&'static mut T` safely [21] [22]. Under the hood it uses atomic checks (and critical sections on no-atomic targets) to ensure you only initialize once. We can leverage this to create a static mailbox without user `unsafe`. For example:

```rust
use static_cell::StaticCell;
use heapless::spsc::Queue;

static MAILBOX: StaticCell<Queue<MyMsg, 8>> = StaticCell::new();

// During startup (before any concurrent access):
let queue_ref: &'static mut Queue<MyMsg, 8> = MAILBOX.init(Queue::new());
let (producer, consumer) = queue_ref.split();
```

In this snippet, `MAILBOX.init(Queue::new())` does an atomic check and then yields a `&'static mut Queue<MyMsg, 8>` which we immediately split. A second call to `init()` would panic, so it's safe from double-initialization. The end result is similar to using a macro: we obtained static producer/consumer handles without writing `unsafe`. The difference is that we explicitly call an init function in code. This pattern is an example of a **builder** or factory function approach – you could wrap this in your own `Mailbox::new()` function that does the init and split internally, returning the endpoints.

- **Encapsulating queue in a struct:** Another idiomatic pattern is to create a struct that owns the static queue and provides methods to get producers/consumers. For instance:

```rust
struct Mailbox<T, const N: usize> {
    queue: StaticCell<heapless::spsc::Queue<T, N>>,
}

impl<T, const N: usize> Mailbox<T, N> {
    const fn new() -> Self {
        Self { queue: StaticCell::new() }
    }
    /// Initialize the mailbox (once) and get producer and consumer
    fn split(&'static self) -> (heapless::spsc::Producer<'static, T, N>,
                                heapless::spsc::Consumer<'static, T, N>) {
        let q: &'static mut heapless::spsc::Queue<T, N> =
            self.queue.init(heapless::spsc::Queue::new());
        q.split()
    }
}

static MY_MAILBOX: Mailbox<Event, 16> = Mailbox::new();
// ...
let (tx, rx) = MY_MAILBOX.split();
```

Here, `Mailbox::new` is a `const fn` so it can be used in a static initializer. The `split()` method uses the `StaticCell` internally to safely get a `'static mut Queue` and then splits it. We mark `split(&'static self)` – requiring a `'static` reference to the Mailbox – to ensure the mailbox itself is static. This abstraction prevents misuse: the raw queue is never exposed publicly, and the only way to get at it is via the safe `split`. The `Mailbox` struct could even implement traits or have methods to directly send/receive messages, further encapsulating the queue. This approach uses Rust's type system to ensure safety (no aliasing beyond the producer/consumer) and gives the user a clear API.

- **Builder with user-provided storage:** A variation is to allow users to provide the memory storage for the queue explicitly (useful if the user wants to place it in a specific region or make it part of another struct). For example, the `heapless` crate has some APIs (like `heapless::Vec`) where you can provide a backing array. While `heapless::spsc::Queue` doesn't currently allow injecting an existing buffer (it manages its own internal array), one could imagine a builder where the user provides a `&'static mut MaybeUninit<[T; N]>` buffer and the library constructs a queue around it using `unsafe` internally. This is an advanced pattern – essentially, **zero-cost abstraction** where the user controls memory. Crates like `bbqueue` do something similar internally (BBQueue's `BBBuffer` owns a `[u8; N]` buffer inside). If absolute control is needed, an actor framework might expose a *"provide your buffer"* API, but this usually requires `unsafe` to interpret the buffer as a ring, so it's often encapsulated within the library.

- **Traits for mailbox initialization:** We can also hide details behind traits. For instance, an actor framework might have a trait `MailboxProvider` that is implemented by different mailbox types. A default implementation might use a heapless queue internally. The user could choose a mailbox type (perhaps via a type alias or associated type in the Actor) and the framework's initialization process uses that to set up the mailbox. This is more about flexibility (choose different queue implementations) than about static vs dynamic memory, but it's relevant: you might provide both a "static SPSC queue" implementation and (for non-embedded use) a dynamic channel implementation, under the same trait. The user picks one by type, and the framework takes care of constructing it properly.

In summary, **idiomatic designs favor wrapping unsafe static-mutation in safe APIs**. Whether through a crate like `StaticCell`, a dedicated `StaticChannel` struct (as in thingbuf), or your own wrapper type, the idea is to ensure that the user never has to deal with raw `static mut` or unsafely leaking references. Instead, they get a well-defined builder or method to call. This approach yields safer code and a clearer intent (the user sees "oh, I call `Mailbox.split()` to get my endpoints" rather than juggling raw static variables).

Ownership, Memory Placement, and Ergonomic Trade-offs

When designing a mailbox API, there are important trade-offs involving who "owns" the mailbox, how visible the memory is, and overall API ergonomics:

- **Global static vs. encapsulated:** A global `static mut Queue` (as in the heapless example) is simple but not very encapsulated. It's essentially a global variable. This can be acceptable in embedded systems (which often rely on some globals), but it makes reasoning about ownership harder – any code with `unsafe` access to that static could manipulate it. By contrast, frameworks like RTIC encapsulate the static in a resource or within a macro expansion, so it's not globally accessible except through the provided handles. Encapsulation here improves safety (you can't accidentally access the queue without going through the Producer/Consumer API).

- **User control of memory vs. ease of use:** Allowing the user to control memory placement (e.g. which RAM bank or section) is crucial in embedded. If the mailbox is hidden entirely inside library code or a macro with no hooks, the user might not be able to place it in fast RAM or in a non-initialized section for retention, etc. For example, if an actor system automatically allocates mailboxes, the user might wonder *"where is this buffer actually located?"*. On the other hand, requiring the user to supply the memory (like passing in a buffer) increases API complexity and the chance of misuse. A middle ground is to have the user declare the static (with whatever attributes needed) but use a provided safe initializer. For instance, the user could declare `static MYBUF: MaybeUninit<[Event; 16]> = MaybeUninit::uninit();` in a specific section, and then call something like `let (p,c) = Mailbox::attach(&MYBUF);`. This attach function would do an unsafe to treat `MYBUF` as a ring buffer and yield endpoints. This gives full control to the user at the cost of a more complex setup procedure.

- **Safety vs. zero-cost:** Using higher-level abstractions (like `Mutex` or `RefCell`) to avoid unsafe can introduce overhead. For example, one *could* put a `static QUEUE: Mutex<Queue<T,N>>` and then have tasks lock it to push/pop. That avoids any unsafe and makes the static borrow checking trivial (because the Mutex ensures only one accessor at a time), but it **defeats the purpose of a lock-free SPSC queue**. It adds runtime cost (interrupt disabling or atomic locking) on each access. So, the idiomatic approach is to use zero-cost abstractions (no mutex, no heap) even if it means hiding some internal `unsafe`. The `heapless::Queue` itself uses atomic operations (or critical sections on targets without atomics) to allow lock-free concurrency [23] [24]. The user doesn't see that complexity, but it's zero-cost in usage (just a few atomics, which are usually much cheaper than a mutex or critical-section every time).

- **API ergonomics:** There's a balance between magical macros and explicit code. Macros like `make_channel!` are very ergonomic but somewhat "hidden" in what they do – new users might be confused about where the memory lives or why it can only be called once. A well-documented macro or an alternative function-based API can mitigate this. For example, `static_cell` is explicit (you call `init` exactly once), which some might find clearer than a macro that implicitly does the same. In documentation and design, you'd want to clarify that *"this creates a static buffer of size N; do not call it twice"*. Providing both options (for instance, a macro for the common case and a manual API for advanced cases) could be ideal.

- **Mailbox ownership models:** In an actor system, one question is: does each actor own its mailbox (in terms of code structure), or are mailboxes defined externally and injected? If the actor "owns" its mailbox, you might allocate it as part of the actor (e.g., as a field in the actor struct or in a static associated with the actor). This makes the mailbox logically tied to the actor's lifecycle. However, if the actor is not a `static` entity (maybe it's constructed at runtime), you can't easily have a truly static buffer inside it without leaking. Many embedded actor frameworks sidestep this by making actors themselves static singletons or using a global initializer. For instance, you might have `static LOGGER_ACTOR: LoggerActor = LoggerActor::new(); static LOGGER_MB: StaticChannel<LogMsg, 32> = StaticChannel::new();` – then mount the actor with that mailbox. The alternative is passing mailbox handles around, which can be more flexible but also more to configure.

- **Single producer/consumer enforcement:** SPSC by definition means exactly one producer and one consumer. The API should make it hard or impossible to accidentally have multiple producers or consumers. Heapless achieves this by consuming the queue in `split()` – you physically **cannot** clone a `Producer` or `Consumer` (they are not `Clone`), and the only way to get them is through that one call. Some frameworks (like RTIC's channels) allow multiple producers (so they use MPSC under the hood). If you truly only want SPSC, you might choose to not implement `Clone` on the producer, ensuring compile-time that you can't have two producers. The trade-off is flexibility: sometimes multiple tasks might enqueue to one mailbox, which is a different pattern (MPSC). But SPSC has performance and simplicity benefits, especially in **single-threaded or interrupt-driven** scenarios – it avoids needing a mutex or more complex atomic operations for multiple writers.

In short, designing the API requires balancing **control vs. convenience**. Embedded developers value determinism and control (knowing where every byte is, avoiding hidden heap allocations), but they also appreciate when a library saves them from writing tricky unsafe code. The best practice is to expose hooks for control (like letting them place a static or specify capacity) while defaulting to safe abstractions. The user should never have to choose between safety and performance – they should get both via the library's design (for example, a compile-time fixed capacity yields predictability, and no runtime malloc means no unexpected OOM errors [25]).

Case Study: Actor Frameworks in Single-Threaded `no_std` Systems

Let's consider how actor frameworks or message-passing systems handle mailboxes in a typical **single-threaded** embedded context (no OS threads, just a main loop and possibly ISRs):

- **Single-threaded executor (cooperative scheduling):** If your actors run on a single thread (cooperatively scheduled via `async/await` or a simple round-robin), then at **no point will two pieces of code access the mailbox truly concurrently** (except interrupts, which we address separately). In such a scenario, the synchronization requirements are lighter. For instance, *Ector* (which runs on Embassy's single-threaded executor) can use an `async` channel without heavy atomics, because the `await` points ensure only one task is running at a time. The `thingbuf::StaticChannel` example above works in an async context: the producer `send().await` will yield if the channel is full, letting the consumer run. Because it's single-core and tasks interleave, they rely on the **Rust async scheduler** for fairness rather than true parallelism. The channel still uses atomics internally to coordinate, but in a single-thread, no-ISR scenario, those could potentially be replaced by simpler checks (still, using the battle-tested lock-free algorithm is fine and saves having different code paths). The key point: **no std, single thread doesn't mean no concurrency** – it just means concurrency is *time-sliced*. An actor model built on a single thread can safely use a lock-free SPSC queue to communicate between tasks (one task plays the "producer" role, another the "consumer").

- **Interrupt handlers as actors:** In many embedded cases, an ISR might produce messages to a queue while a background task consumes them. This is a classic use for an SPSC queue: ISR enqueues data, main loop dequeues. Because an interrupt can preempt the main loop, this *is* concurrent access (even on single core). That's why queues like heapless `spsc::Queue` disable interrupts or use atomics internally to prevent race conditions when one side is in an ISR [26]. Actor frameworks that allow an ISR to send a message must ensure the mailbox is **accessible from interrupt context** and is lock-free. RTIC, for example, would treat the producer as a resource at an interrupt priority and the consumer at task level; the fact that the queue is lock-free means the ISR can push without needing to lock a mutex (important for latency). So in single-threaded *with* interrupts, SPSC queues shine: you avoid needing a critical-section around the whole enqueue operation (the atomic CAS does the job with minimal delay).

- **Other frameworks:** Some embedded messaging systems avoid explicit queues by using *direct calls or state machines* (e.g., sending signals that something is ready). But if using an actor model, a queue per actor is typical. For example, **Hubris** (an embedded OS by Oxide Computer) uses a static message queue for each process/actor (statically allocated in a dedicated memory section), albeit that's at an OS level. In pure Rust, frameworks like **Minimq** (for MQTT) or others often integrate heapless queues for message passing. The patterns remain the same: static allocation, fixed capacities, and no user unsafe code.

- **MPMC vs SPSC trade in single-thread:** An MPMC (multi-producer, multi-consumer) channel is more general but heavier (needs more synchronization). Many embedded use cases really only need SPSC: each task/actor has one incoming message queue (single consumer), and any number of other tasks or ISRs might send to it. If you truly need multiple producers, you either accept an MPSC (with a slight performance hit and more complexity) or you funnel messages through one producer task. Since the question focuses on SPSC, the implication is we expect one producer per mailbox (often the producer is an ISR or a single sending task). Designing the system with SPSC in mind can reduce overhead: for instance, you can avoid a compare-and-swap loop on the enqueue if you **know** only one writer exists (you could use a simple `core::sync::atomic::AtomicUsize` index increment without CAS). Heapless's queue still uses a CAS by default (to be general and because on some architectures it's required), but on Cortex-M, single-producer single-consumer can sometimes be done with just load/store (if using proper memory ordering). Some libraries have compile-time options to optimize for true SPSC. For example, if you have a single-thread executor with no interrupts, you might not need any atomic operations at all – a plain ring buffer with check-and-wait could suffice (since context switches only happen at `await` points). However, using the well-tested lock-free implementations is usually preferable unless you *really* need to squeeze cycles, and they're typically "wait-free" (bounded time operations) which is great for real-time systems [27].

Real-world best practice: Production embedded projects (e.g. drivers, wireless stacks) frequently use **heapless SPSC queues or BBQueue** for passing data between ISRs and thread contexts. They define these queues as `static` globals in a specific memory region (if needed for DMA) and often provide safe API functions to use them. For instance, a UART interrupt might enqueue received bytes into a static queue, and a background task periodically dequeues and processes them. The queue provides backpressure (if full, maybe drop or signal overflow). By using a fixed-size queue, they ensure **deterministic memory usage and time** – no unbounded allocations, and operations are O(1) worst-case [27] .

To conclude this section: **single-threaded embedded actors benefit from static SPSC mailboxes** by achieving concurrency without an OS. The patterns from RTIC, Embassy/Ector, etc., all reinforce using static, capacity-bounded queues for message passing. They hide the unsafety and provide a clean API, so the developer can focus on the actor logic rather than memory management. SPSC is simpler and more predictable than more general channels, aligning with embedded ethos (small and predictable code).

Best Practices and Examples

Finally, let's summarize best practices and provide concrete examples combining the ideas above:

- **Use const generic capacity and inline buffers:** By making the mailbox capacity a compile-time constant, you get a zero-cost, in-place buffer. Crates like heapless and thingbuf follow this pattern. Always choose a capacity that suits your worst-case message load and document that messages will be dropped or overwritten if the queue is full (depending on your policy).

- **Hide `unsafe` inside library code:** If you are writing your own actor framework or mailbox utility, ensure that any `unsafe` needed for `static mut` conversion or pointer casting is done in one place and thoroughly reviewed/tested. Provide a safe API (functions, macros, or structs) so the end user never needs to write `unsafe` for common use. For instance, if using heapless, you might write:

```
// internal function within library
fn split_static_queue<T, const N: usize>(
    q: &'static mut heapless::spsc::Queue<T, N>
) -> (heapless::spsc::Producer<'static, T, N>,
heapless::spsc::Consumer<'static, T, N>) {
    // This unsafe is okay because we ensure exclusive access here.
    // We immediately split and return ownership to caller.
    unsafe { (*q as *mut _).split() }
}
```

The above is a conceptual sketch – in reality `Queue::split()` is a safe method that takes `&mut self`, so you wouldn't need unsafe to call it; the unsafe would be only in obtaining the `&'static mut` in the first place. Either way, limit unsafety to one well-tested spot.

- **Provide macro or static initializer for ergonomics:** For example, using the earlier `static_mailbox!` macro idea:

```
// Example of using a macro to set up a static mailbox
let (logger_tx, logger_rx) = static_mailbox!(LOGGER_Q: LogMessage, 32);
```

This might expand into a static queue and yield a `Producer`/`Consumer`. It dramatically simplifies user code for common cases. In documentation, show this usage for the typical scenario of one mailbox per actor. You can accompany it with a manual alternative using `StaticCell` or similar for advanced users.

- **Avoid global mutable state beyond what's needed:** Even though the mailbox is static, the only mutable state accessible should be through controlled handles. For instance, after splitting a heapless `Queue`, the original `Queue` object ideally is not used directly anymore (in fact, heapless's split consumes the original queue reference). This prevents accidental simultaneous access. If using a `StaticChannel` (thingbuf), once you call `split()`, you should use the `StaticSender`/ `StaticReceiver` and not call `split()` again or try to use the channel object in any other way. Essentially, design the API so that once the mailbox is "running," only message send/recv operations are available, not any low-level buffer manipulation.

- **Document memory placement options:** If your library uses a macro or implicit static, explain where the memory goes (e.g., "will be in .bss by default"). If users need control, either allow an attribute or suggest they manually declare a `static` and use a provided init function. For instance, the **Cortex-M** `cortex_m::singleton!` **macro** is a known pattern to obtain static memory safely. It uses a similar approach with `StaticCell` internally. A user could do:

```
let queue_ref: &'static mut Queue<Msg, 16> =
    cortex_m::singleton!(: Queue<Msg, 16> = Queue::new()).unwrap();
let (tx, rx) = queue_ref.split();
```

This uses a macro from the `cortex_m` crate to create a static memory location and return a mutable reference to it [28] [29]. It's another option if one is already in a Cortex-M environment. The upside is it's proven and simple; the downside is it's Cortex-M specific and requires the `cortex_m` crate.

- **Test in a single-thread context:** If your mailbox is meant for single-thread + ISR usage, test that scenario. For example, enqueue from an interrupt (which in tests might be simulated by calling an "ISR" function that enqueues) while dequeuing from a loop. Ensure no races or data corruption. The heapless queue and thingbuf channel have been tested in such scenarios (they handle the necessary memory fences [30]), but if you roll your own, be careful with memory order (e.g., use `core::sync::atomic::compiler_fence` or appropriate `Ordering` on atomic ops so that the producer and consumer see memory consistently).

- **Evaluate message ownership semantics:** In some actor models, you pass around owned messages (e.g., move the struct into the queue). In others, you might use references or pools to avoid copies. A static mailbox implies the data lives in the queue's buffer. For large messages or buffers, consider using a memory pool (the `heapless::pool` module or similar) and send pool indexes through the mailbox. That is beyond the scope of the question, but worth noting as an advanced pattern – you still allocate everything statically, but indirectly.

To wrap up, **embedded Rust makes it possible to have fully static, zero-heap, zero-copy communication channels that are memory-safe**. The combination of `const` generics, `static` variables, and a dash of macros/unsafe (encapsulated) yields a powerful recipe for actor message passing on microcontrollers. The key is to leverage community-tested crates (heapless, static_cell, etc.) and patterns (RTIC's approach, Embassy's async channels) to guide the design. By doing so, you can provide an actor system where developers explicitly place their mailboxes in memory, know their exact capacity, and never have to worry about a stray pointer or `unsafe` mistake corrupting their system. All message passing will be through well-defined, compile-checked interfaces (e.g., a `Producer<Msg>` that can only enqueue `Msg` types, etc.), leading to an **ergonomic and safe developer experience**.

Concrete Example: Macro vs Manual Setup

To illustrate, here are two ways one might set up a static SPSC mailbox for an actor handling `SensorEvent` messages with capacity 10:

Using a helper macro (hypothetical):

```
// Define a static mailbox for SensorEvent with 10-slot queue
let (sensor_tx, sensor_rx) = static_mailbox!(SENSOR_Q: SensorEvent, 10);
// sensor_tx: Producer<'static, SensorEvent, 10>
// sensor_rx: Consumer<'static, SensorEvent, 10>

// Now sensor_tx can be given to an interrupt or another task,
// and sensor_rx given to the SensorActor for processing.
sensor_tx.enqueue(SensorEvent::Reading(42)).unwrap();
```

Everything here is safe. If `static_mailbox!` is implemented as discussed, it handles the static definition and split internally. The user just gets the resulting handle and uses it.

Using `StaticCell` *manually:* (no macro, but explicit)

```
static SENSOR_QUEUE: StaticCell<heapless::spsc::Queue<SensorEvent, 10>> =
StaticCell::new();

// ... in initialization code
let queue_ref: &'static mut heapless::spsc::Queue<SensorEvent, 10>
    = SENSOR_QUEUE.init(heapless::spsc::Queue::new());
let (sensor_tx, sensor_rx) = queue_ref.split();
// sensor_tx and sensor_rx are now ready to use.
```

This code achieves the same outcome. The `StaticCell` ensures `queue_ref` is a unique `'static mut` reference. After the split, you would typically not use `queue_ref` directly anymore (and you **must not call** `SENSOR_QUEUE.init()` again or it will panic at runtime [31]). The producer and consumer can then be passed where needed.

Both approaches have zero runtime overhead beyond the ring buffer operations themselves. They avoid the heap entirely and do not require any dynamic memory. The choice between them might come down to how much abstraction you want – macros can reduce boilerplate in user code, whereas the manual approach might be a bit more transparent.

Final words: The best pattern is one that **ensures safety, fits the embedded constraints, and is ergonomic enough that developers won't be tempted to improvise their own unsafe solutions**. The patterns discussed – from RTIC's safe static resources, to `heapless::Queue` + `StaticCell`, to dedicated static channel types – all achieve this in slightly different ways. Any of them can be considered idiomatic in modern Rust. The common thread is that the user retains control over placement and capacity, and the library takes care of the unsafe heavy lifting. By following these patterns, you can implement actor mailboxes on tiny no_std systems that are as robust as those on a full OS, with the confidence of Rust's safety guarantees.

Sources:

- Heapless SPSC queue usage and `'static mut` pattern [3] [32]
- RTIC example using a local static queue split into producer/consumer [6] [8]
- Drogue IoT example with static mut queues for ISR communication [16] [33]
- RTIC's `make_channel!` macro (ensuring one-time split) [19] [20]
- Thingbuf static channel example (const init and split) [11] [12]
- `StaticCell` crate for safe static mut initialization [21] [22]
- RTFM v0.3 (early RTIC) blog discussing `'static mut` for ring buffers [28] [29] (motivation for these patterns)
- Heapless crate documentation on static data structures and no-alloc behavior [1] [2]
- Real-time considerations of heapless (capacity and time determinism) [27] and RTIC channel usage [17] [26].

---

1 2 25 27 heapless - Rust
https://rtic.rs/2/api/heapless/index.html

3 4 5 23 24 30 32 heapless::spsc - Rust
https://docs.rs/heapless/latest/heapless/spsc/index.html

6 7 8 'static super-powers - Real-Time Interrupt-driven Concurrency
https://rtic.rs/1/book/en/by-example/tips_static_lifetimes.html

9 10 thingbuf - Rust
https://docs.rs/thingbuf/latest/thingbuf/

11 12 thingbuf::mpsc - Rust
https://docs.rs/thingbuf/latest/thingbuf/mpsc/index.html

13 14 15 GitHub - jamesmunns/bbqueue: A SPSC, lockless, no_std, thread safe, queue, based on BipBuffers
https://github.com/jamesmunns/bbqueue

16 33 WiFi Offloading — Drogue IoT
https://blog.drogue.io/wifi-offload/

17 18 26 Channel based communication - Real-Time Interrupt-driven Concurrency
https://rtic.rs/2/book/en/by-example/channel.html

19 20 channel.rs - source
https://docs.rs/rtic-sync/latest/src/rtic_sync/channel.rs.html

21 22 31 static_cell - Rust
https://docs.rs/static_cell

28 29 RTFM v0.3.0: safe `&'static mut T` and less locks | Embedded in Rust
https://blog.japaric.io/rtfm-v3/