ChatGPT

Safe, Idiomatic Static Mailbox Allocation in no_std Rust Actor Systems

Safe Patterns for Static SPSC Queues Without `unsafe`

Achieving a static, fixed-capacity mailbox **without any `unsafe` code** is possible using modern Rust patterns. One key approach is to avoid `static mut` and instead leverage safe abstractions that provide `'static` references at runtime:

- **Use a `StaticCell`** (from the [`static_cell` crate]): This is a no_std, no-alloc container that lets you reserve memory at compile-time and initialize it later, returning a `&'static mut T` safely. For example:

```
use static_cell::StaticCell;
static MAILBOX_BUF: StaticCell<heapless::spsc::Queue<Msg, N>> =
StaticCell::new();

// During initialization (e.g., in main):
let queue: &'static mut heapless::spsc::Queue<Msg, N> =
MAILBOX_BUF.init(heapless::spsc::Queue::new());
let (producer, consumer) = queue.split();
```

Here, `StaticCell::init` returns a `&'static mut` safely (panicking on double initialization) [1]. The returned `producer` and `consumer` have a `'static` lifetime because they borrow the static queue.

- **One-Time Initialization with `OnceCell` or `Lazy`:** If you only need an immutable static reference (`&'static T`), a `OnceCell`/`OnceLock` can initialize data at runtime. However, for mutable access or no_std contexts, `OnceCell` in core is not thread-safe and doesn't cover `&'static mut` [2]. `StaticCell` is the specialized solution here.

- **Builder or Factory APIs:** Provide an API where the user *constructs* the queue at runtime (e.g., in an init function) and obtains the producer/consumer without manual unsafe. For instance, a `MailboxBuilder` could internally hold a `MaybeUninit<Queue>` and on `.build()` do the initialization and `split()`, returning the endpoints. Internally this might use `unsafe`, but the user-facing API remains safe. The key is to perform initialization exactly once and never allow a second split (to avoid aliasing). Tools like `MaybeUninit` plus careful tracking can enforce this.

- **Leverage `const fn` for initializers:** The `heapless::spsc::Queue` provides a `const fn new() -> Self` [3], so it *can* be placed in a `static`. The tricky part is splitting it safely. If possible, prefer types that offer a safe split via interior mutability. If you control the type, you might design it similarly to other libraries (see below) to allow safe static usage.

Patterns in Existing Embedded Libraries

Several embedded Rust libraries have *idiomatic solutions* for static SPSC queues or channels that avoid user-level unsafe:

- **BBQueue's `BBBuffer`:** The [`bbqueue`] crate offers a lock-free SPSC queue designed for DMA and embedded use. It allows static allocation and *safe* splitting via an internal mechanism. For example, one can declare `static BB: BBBuffer<6> = BBBuffer::new();` and then in `main` do `let (prod, cons) = BB.try_split().unwrap();` [4]. The `try_split` method uses interior mutability (atomics/critical sections) so that it only splits once – any second attempt returns an error [5]. This ensures that you cannot accidentally create two producers or two consumers, maintaining safety without requiring the user to write `unsafe`. The `BBBuffer` design is entirely safe to use with `#![deny(unsafe_code)]` in your crate (any necessary `unsafe` is confined within `bbqueue`'s implementation, not your code).

- **Thingbuf's `StaticChannel`:** The [`thingbuf`] crate provides asynchronous MPSC channels and includes a static, fixed-size flavor for no_std. It has a `StaticChannel<T, N>` that can be declared in a `static` initializer and then safely split. For example, you can do:

```
static KERNEL_EVENTS: thingbuf::mpsc::StaticChannel<KernelEvent, 256>
    = thingbuf::mpsc::StaticChannel::new();

// ... later in code:
let (event_tx, event_rx) = KERNEL_EVENTS.split();
```

This yields a `StaticSender` and `StaticReceiver` safely [6]. Internally, `StaticChannel` ensures the backing array is in static memory and that splitting is done only once. The design mirrors the approach of BBQueue – the user never handles raw pointers or `unsafe`. The above snippet shows a fully static allocation in a `#![no_std]` context with no heap [7] [8].

- **Embassy's StaticCell and Channels:** The Embassy async framework encourages using `StaticCell` for executors and channels. Embassy's documentation notes that to obtain a `&'static mut` executor or channel, **"a StaticCell (safe) [is one way]; a static mut (unsafe) [is another]"** [9]. Embassy provides channel implementations (like `embassy::channel::Channel`) that can be statically allocated using `StaticCell` or const initializers. The overall pattern is to use a safe cell or wrapper instead of direct `static mut`.

- **RTIC (Real-Time Interrupt-driven Concurrency):** RTIC's framework macros allow static resources without user unsafe. In RTIC 0.5+, if you declare a resource in the `#[init]` function's `local` variables, the framework grants it `'static` lifetime safely [10] [11]. For example, you can have:

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

RTIC transforms this under the hood into a `static` and ensures the queue is not accessed concurrently in unsound ways. The outcome is that two tasks can share a `heapless::spsc::Queue` for lock-free communication **without any explicit unsafe** [12] [13]. In fact, RTIC is designed such that *"shared resources [have] safe accesses without the use of unsafe code."* [14]. This is achieved by the framework restricting when and how each resource is accessed (e.g., one task gets the producer, another the consumer).

- **Cortex-M `singleton!` macro (if using ARM Cortex-M):** Although not a general library, it's worth noting the `cortex_m::singleton!` macro as a pattern. It uses linker tricks or static memory behind the scenes to allocate data and returns a `&'static mut T` safely (panicking on second initialization). This macro was inspired by RTIC's approach [15]. It allows a static buffer to be obtained safely in a one-time init context. If your platform allows it, this is another route to avoid writing unsafe in your code.

Takeaway: Many libraries solve static allocation by introducing a layer that performs any necessary unsafe internally, exposing a safe API (e.g., `split()` or `try_split()`) that enforces correct usage. These designs often rely on *interior mutability*, atomics, or one-time init patterns to prevent misuse like double allocation or aliasing. Adopting a similar strategy in your actor framework – either by using such a crate or emulating the pattern – will let you meet the `#![deny(unsafe_code)]` requirement.

Stable Alternatives to `unsafe` Static Splitting

If you want to continue using `heapless::spsc::Queue` but avoid `unsafe`, consider these strategies:

- **Wrap the Static Queue in a Safer API:** Instead of exposing an `unsafe fn create_mailbox()` that does `&mut *addr_of_mut!(static_queue)`, wrap the static queue in a module or struct that ensures safe access. For example, a struct could hold a `MaybeUninit<Queue<T,N>>` and track whether it's been initialized and split. On first use, you call a safe `init_mailbox()` which does the `Queue::new()` and `split()`. Subsequent calls can be disallowed (panic or Error) to avoid re-splitting. This pattern is essentially what `StaticCell` or `OnceCell` provides – you can implement it yourself with care, or simply use those crates to save effort [2].

- **Use Const Generics and Array Storage Directly:** In some cases, you might allocate a static array for the buffer and manage indices yourself with atomic operations or a `RefCell`. This is more involved, but crates like `fring` or a custom minimal ring buffer can avoid unsafe by construction. However, since `heapless::Queue` is already well-tested, a safer wrapper or cell around it is the simpler approach.

- **No-Allocator, No-std Considerations:** All the above approaches work in no_std without `alloc`. `StaticCell` is no_std compatible [16] and uses only core/atomic primitives (on targets without atomics it uses a `critical-section` crate fallback [17]). Thus, you don't need a global allocator; you're still doing static, fixed-size allocation – just in a safer way.

In summary, there **are** zero-unsafe alternatives for splitting a static SPSC queue: - Use a safe one-time initialization primitive (like `StaticCell`) to get a `&'static mut Queue` without `unsafe`. - Or choose a different SPSC queue implementation (BBQueue, ThingBuf) that is designed for safe static splitting. - Or restructure your API to initialize and split the queue in a controlled, single spot (e.g., during system startup) so that Rust's borrow rules are satisfied without explicit unsafe.

Each of these preserves your goals: no dynamic allocation, no_std compatibility, and essentially zero runtime overhead (these patterns incur at most one-time checks or atomic flags, which are negligible).

Isolating and Documenting Unsafe Code (if Truly Unavoidable)

If after exploring the above you find that some `unsafe` is still necessary (for example, due to constraints of your API or to avoid dependence on another crate), the next best practice is to **isolate and clearly mark** that unsafe code:

- **Feature-gate the Unsafe Implementation:** Continue using `#![deny(unsafe_code)]` by default, but put any unsafe operations behind an opt-in feature flag (e.g., `"unsafe_opt"`). You can do this by conditionally compiling a module or function only when the feature is enabled. For instance:

```
#[cfg(feature = "unsafe_opt")]
pub unsafe fn create_mailbox() -> (Producer<'static, T, N>, Consumer<'static,
T, N>) { … }
```

The crate root could include `#![cfg_attr(not(feature = "unsafe_opt"), deny(unsafe_code))]` so that enabling the feature lifts the `deny` for that compilation. This ensures the CI (which runs with default features) sees zero unsafe, satisfying tools like cargo-geiger. Only users who explicitly enable `"unsafe_opt"` will compile the unsafe code.

- **Small, Auditable Unsafe Regions:** Keep the `unsafe` as localized as possible. For example, just one small `unsafe { ... }` block where you convert a static buffer to a `&'static mut Queue` (or call `addr_of_mut!` as in your current code) is easier to verify than widespread unsafety. All other code (like actually enqueueing/dequeueing) can remain safe. By minimizing the unsafe surface, you reduce the risk of unsoundness.

- **Document Invariants with SAFETY Comments:** Every `unsafe` block or function should have a **SAFETY:** comment explaining why it's sound. For instance, if you do `static mut QUEUE: Queue<T,N> = Queue::new();` and later `unsafe { &mut QUEUE }`, document that "*SAFETY: This static is only accessed here during initialization and then split into producer/consumer halves given to distinct contexts. After splitting, no two mutable references exist: producer and consumer use disjoint indices internally* [5]*. We ensure this init happens before any concurrent access (e.g., before starting interrupts or multi-core tasks).*" Be explicit about single-threaded context or interrupt masking that makes the usage safe. Well-explained invariants build trust for those reviewing the code.

- **Use `#[cfg_attr]` for Docs and Enforcement:** Mark unsafe APIs with `#[cfg(feature = "unsafe_opt")]` and use `#[cfg_attr(docsrs, doc(cfg(feature = "unsafe_opt")))]` so that documentation clearly shows these functions require the feature. This prevents accidental usage. Also, structuring your code with separate modules (e.g., a module `mailbox_unsafe` that is only compiled with the feature) can make it obvious which parts are unsafe. At the API level, consider naming such functions `unchecked_...` or marking them `unsafe fn` if the caller must uphold certain conditions.

- **CI Strategies:** Your CI should test both with and without the feature. By default (feature off), `cargo geiger` and `deny(unsafe_code)` will assert there's no unsafe. Additionally, consider a CI job that builds with the feature *on* (and perhaps with `RUSTFLAGS="-D warnings"` but **without** `deny(unsafe_code)` so it can compile) to ensure the unsafe-enabled code doesn't bit-rot and actually remains sound. This two-pronged approach catches issues early.

- **Follow the Footsteps of Others:** Many projects isolate unsafe this way. For example, some crates have an `"unsafe_atomic"` or `"unsafe_unchecked"` feature for performance tweaks. By default they run safely, but advanced users can opt into an unsafe path if they need absolute zero-cost (and are willing to trust the invariants). If you do this, **loudly advertise** in documentation that the feature enables internal unsafe code and why one might (or might not) want to use it.

In conclusion, prefer a **safe abstraction approach** if at all possible (using something like `StaticCell` or a safe split as in BBQueue/ThingBuf). This will satisfy `deny(unsafe_code)` and keep your library ergonomic. If you must use unsafe for static mailboxes, contain it behind a feature gate and thoroughly document it. By examining how frameworks like RTIC and Embassy handle static data, we see that it's feasible to achieve zero-unsafe patterns in embedded Rust [9] [13]. Adopting those idioms will make your actor system sound, **CI-friendly**, and maintainable.

Sources:

- Heapless SPSC Queue documentation [18] [19]
- Embassy Executor docs (using `StaticCell` vs `static mut`) [9]
- *static_cell* crate example (`StaticCell::init` returning `&'static mut`) [1]
- BBQueue documentation (safe static `BBBuffer` and one-time `try_split`) [4] [5]
- Thingbuf documentation (using `StaticChannel` with `split()` in no_std) [6]
- RTIC example (splitting a heapless Queue in `#[init]` safely with `'static` lifetime) [11] [13]
- User discussion on RTIC's safe transform of `static mut` (inspiration for `cortex_m::singleton!`) [15].

---

[1] [2] [16] [17] static_cell - Rust
https://docs.rs/static_cell/latest/static_cell/

[3] [18] [19] Queue in heapless::spsc - Rust
https://rtic.rs/stable/api/heapless/spsc/struct.Queue.html

[4] [5] bbqueue - Rust
https://docs.rs/bbqueue/latest/bbqueue/

6 7 8 thingbuf::mpsc - Rust
https://docs.rs/thingbuf/latest/thingbuf/mpsc/index.html

9 Executor in embassy_executor - Rust
https://docs.embassy.dev/embassy-executor/git/std/struct.Executor.html

10 11 12 13 'static super-powers - Real-Time Interrupt-driven Concurrency
https://rtic.rs/1/book/en/by-example/tips_static_lifetimes.html

14 Resources - Real-Time Interrupt-driven Concurrency - RTIC
https://rtic.rs/1/book/en/by-example/resources.html

15 Rtic v0.5 complains about static mut variable - Rust Users Forum
https://users.rust-lang.org/t/rtic-v0-5-complains-about-static-mut-variable/67550