ChatGPT

Implementing a Static, Zero-Allocation SPSC Queue in Rust (No `unsafe`, No Heap)

Overview of the Challenge

Implementing a **single-producer, single-consumer (SPSC) queue** in Rust with a `'static` lifetime, no heap allocation, and no `unsafe` code is challenging but achievable. The goal is a lock-free, **zero-allocation** FIFO queue that works in both `std` and `no_std` contexts, suitable as an actor mailbox. The `'static` lifetime requirement means the queue's memory must live for the entire program (or be safely leaked), which typically implies using global/static storage or arenas. However, using **global statics** directly can introduce unsafety (e.g. `static mut`) or require heap allocations (`Box::leak`), both of which we must avoid. We need to leverage **const generics** and compile-time allocation, along with safe abstractions for static initialization, to meet these constraints.

In summary, the solution will rely on *statically allocated ring buffers* (bounded circular buffers) implemented with const generic backing arrays (no heap), and use existing safe patterns (or crates) to obtain `'static` references without writing `unsafe` code. Below we explore how other projects achieve this and outline best practices and available crates for a **zero-alloc, `unsafe`-free static SPSC queue**.

Patterns for Static SPSC Queues in No-Std (Without Unsafe)

Embedded Rust and other no-std projects commonly use **statically allocated ring buffers** for communication between interrupt handlers, tasks, or actor-like components. A classic approach is using a fixed-size array as the buffer and two index counters (head/tail) for the producer and consumer. In Rust, crates like `heapless` provide exactly this: a statically allocated SPSC queue using const generics. The `heapless::spsc::Queue<T, N>` type allocates an array of type `T` with capacity `N` (actually stores up to `N-1` elements to distinguish full vs empty) internally, avoiding any runtime allocation [1]. Its API offers lock-free `enqueue()` and `dequeue()` operations, and it has a `const fn new()` so it can be constructed at compile time [2]. This makes it well-suited for no_std usage. In fact, the RTIC framework's documentation uses `heapless::spsc::Queue` as an example of a *"splittable"* data structure that can be shared between a producer and consumer task in an embedded context [3] [4].

RTIC (Real-Time Interrupt-driven Concurrency) demonstrates a safe pattern for static SPSC queues using its framework: you can declare a queue as a `static` resource (with const-initialization) and then **split** it into producer and consumer halves. For example, in an RTIC `#[init]` function one might do:

```
#[init(local = [q: Queue<u32, 5> =
Queue::new()])]   // static queue of capacity 5
fn init(cx: init::Context) -> (..., Local, ...) {
    // The queue `q` has 'static lifetime, so it remains valid after init
```

```
returns
    let (p, c) = cx.local.q.split();
    // p: Producer<'static, u32, 5>, c: Consumer<'static, u32, 5>
    ...
}
```

Here, the RTIC framework ensures `q` is a static allocation (backed by `[u32;5]` internally) with `'static` lifetime, and the `split()` method yields a producer and consumer that each have `&'static` references to the underlying queue [4]. This is all done without user `unsafe` code – the RTIC framework and `heapless` crate handle the internals safely. This pattern shows that **other no_std projects achieve static SPSC queues by using const generic ring buffer structures and splitting them at runtime, often with the help of a framework or macro to place them in static memory.** The key is that the queue is fully determined at compile-time (capacity and type fixed), so no heap is needed, and the safe API prevents misuse.

Without a framework like RTIC, you can still apply the same idea manually: declare a static ring buffer and split it. However, raw static mutables in Rust require unsafe access. For example, doing `static mut Q: Queue<T,N> = Queue::new();` would allocate the queue statically, but any access to `Q` would be `unsafe` because of potential data races. We need a safe way to get a mutable `'static` reference to a data structure initialized at runtime. This is where crates like `static_cell` come into play.

Safe Static Initialization with `StaticCell`

The `static_cell` crate provides a safe abstraction for static memory that can be initialized at runtime. It essentially wraps a `static mut MaybeUninit<T>` pattern inside a safe API. You declare a `StaticCell<T>` as a `static` (which reserves uninitialized memory for a `T`), and then at runtime call `.init(value)` exactly once to place a value into it, receiving a `&'static mut T` back [5] [6]. If `.init()` is called more than once it will panic, which prevents double initialization. Under the hood it uses atomic operations (via the `portable-atomic` crate) to ensure thread-safe one-time initialization even in no_std environments [7]. Importantly, **you don't write any `unsafe` yourself** – `StaticCell` handles the memory operations safely.

Using `StaticCell`, we can create a static SPSC queue as follows:

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

static QUEUE_MEM: StaticCell<Queue<u32, 16>> = StaticCell::new();  // reserve
space for a Queue

// ... later, e.g. during initialization:
let queue: &'static mut Queue<u32, 16> = QUEUE_MEM.init(Queue::new());
let (producer, consumer) = queue.split();
```

This pattern gives a `'static` queue safely: the `QUEUE_MEM` is a true static, but we access it through the `init` API which ensures it's initialized exactly once and returns a safe `'static mut` reference. The resulting `producer` and `consumer` can be stored or passed to different parts of the system (e.g. an actor's inbox/outbox handles), satisfying the `'static` lifetime requirement. The lit-bit project itself adopted this approach via a helper macro `static_mailbox!` which under the hood uses `StaticCell::init()` to replace a previous unsafe static init solution [8]. After this change, their actor mailbox creation uses no unsafe code while still being zero-alloc and no_std compatible [9].

Best practices for `StaticCell`: You typically declare a `StaticCell<YourType>` as a `static` (possibly public or in a module where needed), and give it an explicit type (including any const generic parameters). At runtime (during system setup or actor spawn), call `.init(...)` with the value (e.g. `Queue::new()` or any constructor). Once initialized, you use the returned `'static mut` reference normally. Remember that `StaticCell` requires a `critical-section` implementation on targets without atomic CAS, as noted in its docs [7] – in other words, on bare-metal targets it will use a critical section to emulate the atomic needed for one-time init. This is usually fine (just ensure you have a `cortex_m`-based or other critical-section crate enabled if on no_std bare metal).

Using `StaticCell` (or analogous patterns like `cortex_m::singleton!` macro for ARM Cortex-M) is a **common pattern in embedded Rust** to get `'static` references without unsafe. Alternatives include `OnceCell` / `OnceLock` from the standard library, but those either require `std` or don't give `&'static mut`. `StaticCell` was designed exactly for scenarios like this [10]. If an allocator were available, one could use `Box::leak(Box::new(...))` to get a `'static mut` reference, but that uses the heap (disallowed here) [10]. And if you were willing to use unsafe, you could do `static mut BUF: MaybeUninit<Queue<T,N>> = MaybeUninit::uninit()` and later `BUF.as_mut_ptr().write(Queue::new())`, but the whole point is to avoid explicit unsafe. So `StaticCell` is a superior choice that meets our constraints.

Generic static containers: One challenge the team faced was storing **generic** queue types in a static. Rust requires the exact type of any static to be known. This means you often need to declare a separate `StaticCell<Queue<MessageType, N>>` for each message type (or use a macro to generate them). In lit-bit, they introduced a macro like `static_mailbox!(Name, Type, Capacity)` to declare and initialize such statics conveniently. This is a reasonable approach: use a macro to allocate a `StaticCell<Queue<T,N>>` and perhaps even call init/split, hiding the verbosity. The bottom line is that other projects have successfully created static SPSC queues by combining const generic ring buffer types (like `heapless::Queue`) with safe static init patterns (like `StaticCell` or RTIC resources), **completely avoiding heap and unsafe code**.

Alternatives to `heapless` for Zero-Allocation SPSC Queues

Your current no_std solution uses `heapless::spsc::Queue`, which is a solid choice (it's widely used in embedded). However, there are alternative crates and implementations that also fulfill the requirements, each with its own features:

- `heapless::spsc::Queue` – *Baseline solution.* Provides a lock-free ring buffer (capacity `N-1` elements) using an array `[T; N]` internally [1]. It is no_std, requires no allocator, and uses only atomic/critical-section for synchronization. Operations are O(1). It's battle-tested in many embedded

projects and used in frameworks like RTIC. One note: by default `heapless` uses the `core::sync::atomic` APIs which on some targets may need the `portable-atomic` feature if the target lacks atomic support (heapless has features to enable a critical-section based fallback). In short, heapless is **production-proven** for embedded SPSC queues.

- **BBQueue (`bbqueue` crate)** – A SPSC queue based on a bipartite buffer (bip-buffer) design by James Munns et al. [11]. BBQueue is also no_std and uses const generics for capacity. It differs from heapless in that it hands out contiguous **grants** of the buffer for writing/reading, which is very useful in DMA or streaming scenarios (you can write a whole chunk and then commit it). It's lock-free and thread-safe. For example, you can declare `static BB: BBBuffer<256> = BBBuffer::new();` and then at runtime do `let (mut prod, mut cons) = BB.try_split().unwrap();` to get producer/consumer handles [12]. These handles can be sent to different threads or an interrupt context, as BBQueue ensures proper SPSC safety [13]. BBQueue is used in real-time audio and embedded systems and comes with a thorough design explanation (Ferrous Systems blog post) [14]. It does use some `unsafe` internally for the buffer management, but the public API is safe. Capacity must be a power of two (internally, it aligns to the nearest power of two). BBQueue is a **mature, performant alternative** especially if you need to minimize copying (because of its grant-based API).

- **Fixed-Queue (`fixed-queue` crate)** – This crate provides fixed-capacity data structures (Vec, VecDeque, etc.) including an SPSC queue and even MPMC, all using const generics and no heap. It explicitly advertises that all structures are `const` **(static-friendly),** `no_std`**,** `no_alloc`**, lock-free and wait-free** [15]. The SPSC queue in `fixed_queue::spsc` likely functions similarly to heapless (backed by an array `[T; N]`). If you need a different API or want a single crate for multiple container types, `fixed-queue` could be useful. It appears to prioritize being usable in interrupt contexts (wait-free, meaning the operations have a bounded execution time with no blocking). This crate is less famous than heapless, but it's another viable zero-allocation option.

- **Fringe (`fring` crate)** – A newer crate that provides a **fast ring buffer** optimized for no_std [16]. It specifically supports a single producer and single consumer, enforced at compile time for safety, and is completely lock-free at runtime [17]. `fring::Buffer<N>` is the ring buffer type; like others, you split it into a producer and consumer halves (with `buffer.split()` returning two structs). One caveat: `fring` requires the capacity `N` to be a power of two (this simplifies index arithmetic with bitwise wrap-around) [18]. Also, its API for static usage isn't as straightforward – the docs show that if you declare a `static BUFFER: Buffer<N> = Buffer::new()`, obtaining the producer or consumer from that static requires an unsafe call (because the safe `split()` takes &mut self) [19]. In other words, `fring` expects you to own the buffer in a function and then split; for truly static usage you'd need to use something like `StaticCell` or manually ensure only one accessor (the crate provides an `unsafe { BUFFER.producer() }` example [20]). Despite that, `fring` is very lightweight and has minimal overhead (just two `usize` indices plus the array) [21]. If you pair `fring` with `StaticCell`, you could avoid that unsafe; e.g., store the `Buffer<N>` in a StaticCell and init it at runtime, similar to the heapless example above.

- **Others:** There are other crates or patterns (e.g., `async-channel` / `flume` for async, but those use heap internally; `crossbeam-channel` in std, not no_std; `heapless::MPMC` for multi-producer; etc.). For our focus on a static SPSC, the above options are the primary ones. Also note, **Embassy's**

async executor provides `embassy_sync::Channel` which is a static-friendly channel for async tasks (using a `NoopMutex` in single-threaded mode) [22], but it's designed for async/Await usage rather than a simple blocking or polling queue. In an embedded actor system without full async, a simpler ring buffer (like heapless or BBQueue) is usually sufficient and easier to integrate.

All these alternatives are **zero allocation and no_std compatible**, using const generic arrays as storage. They are also **battle-tested in production-like environments** (embedded devices, drivers, etc.). For example, BBQueue is used for deferring DMA data, heapless is used all over embedded projects for sensor data queues, etc. Each internally might use some `unsafe` for performance (which is acceptable since those crates have been vetted), but from the outside they present safe interfaces that align with your project's `#![forbid(unsafe_code)]` requirement (forbidding unsafe in *your* code).

Mailbox Patterns in Embedded Actor Systems

Embedded-focused actor or message-passing frameworks generally **avoid dynamic allocation** and enforce static lifetimes for messages and mailboxes. This is both for memory safety (no heap fragmentation or allocation failures) and for real-time guarantees. For instance, **Drogue Device** (an embedded actor framework) is designed such that *"the system is ultimately* `'static`*-centric"* – all actors and messages must be `'static` [23]. This ensures that their async message handling doesn't run into dangling references. In practice, this means the actor's mailbox (queue) is often a static buffer or located in a static memory region. Drogue's approach was to require actors to implement handlers like `fn on_notify(&'static mut self, msg: M)` [24], essentially forcing the actor state and its messages to live forever (or until program end). While this can be somewhat limiting (and Drogue had to be very careful to avoid unsoundness in how it lent out `&'static mut self` to async tasks), it underscores the point: **embedded actor systems manage mailboxes by pre-allocating them statically.**

If we look at how our `lit-bit` project (or similar designs) handle actor mailboxes across std and no_std: - In a `std` environment, one might use a dynamic channel (like `tokio::sync::mpsc`) which allocates. This provides backpressure by suspending the sender when full, but uses the heap. - In a `no_std` environment, dynamic allocation isn't available, so a fixed capacity queue (like heapless) is used with a *fail-fast* strategy (if full, the message send returns an error immediately) [25]. This is exactly what we had: `tokio::mpsc` for std, and `heapless::spsc::Queue` for no_std [26].

To unify these, embedded actor systems typically *abstract the mailbox behind traits or type aliases*. For example, lit-bit defines type aliases `Inbox<T,N>` and `Outbox<T,N>` for the consumer and producer ends of the queue, and uses conditional compilation to alias them to either the heapless or tokio versions [26]. The **message-passing API** (e.g., an actor's `Address<T>` with a `send()` method) is made agnostic to the underlying queue. This way, the actor code doesn't care if it's using a heapless static queue or a tokio channel – it just calls `send` and handles a possible `SendError`. This abstraction can remain 100% safe if designed properly.

It's worth noting that some embedded actor frameworks forego fancy queues altogether and rely on **cooperative scheduling**: e.g., each actor runs to completion and explicitly yields, processing one message at a time from a simple ring buffer. Frameworks like RTIC achieve "actor-like" behavior by scheduling tasks with posted messages (where the message might just be stored in a static buffer or register). But for a general actor system like lit-bit that wants a mailbox per actor, the approach we've discussed – **a static SPSC**

queue per actor – is standard. The only question is how to allocate that queue's memory. The answer, as we've seen, is to either require the user to provide a static buffer (as parameters or via macro) or to have a static memory pool.

Dynamic actor creation in no_std: If your actor system allows creating new actors at runtime in a no_std context, you cannot allocate new mailboxes on the fly from the heap. One strategy is to allocate from a **pre-defined memory pool** of queue buffers. This can be done with an object pool of `Queue<T,N>` instances or a slab of memory that can be partitioned. However, managing such a pool without unsafe is complex. A simpler approach is to require that all actors (and their mailboxes) are known at compile-time in no_std mode (or at least a maximum number known). Many embedded systems indeed have a fixed set of actors/tasks. If truly dynamic creation is needed, you might consider using a single global queue with multiplexed messages, or implementing a custom allocator that draws from static memory (which reintroduces a form of `unsafe`). Given the constraints, **the safest design is to instantiate all needed SPSC queues up front** (each with `StaticCell` or as static globals), and pass out handles to producers/consumers to whatever code needs them (e.g. on actor startup). This is essentially what your `create_mailbox_safe(&StaticCell<Queue<T,N>>)` function achieves [27] – the user provides a `StaticCell` (static buffer) for the mailbox, and you initialize it and return the in/out handles.

Leveraging Const Generics and Compile-Time Evaluation

Rust's **const generics** and compile-time evaluation are key enablers for our solution. They allow us to create types like `Queue<T, N>` where `N` (capacity) is a compile-time constant, and to have a `const fn new() -> Queue<T,N>`. This means the compiler knows the exact size and layout of the queue (essentially an array of `N` items plus indices) and can allocate it in static memory or on the stack. All the crates mentioned (heapless, BBQueue, fixed-queue, fring) use const generics for capacity. This yields several benefits:

- **No runtime overhead for allocation:** The memory is reserved at compile time. For example, `heapless::Queue::<u8, 32>::new()` will internally include a `[MaybeUninit<u8>; 32]` (or 33 to differentiate full/empty) as part of its struct. No `Vec` or heap allocation occurs at runtime; you pay only the fixed memory cost. This is perfect for embedded and real-time systems.

- **Const `new()` allows static initialization:** Because `new()` is a `const fn` [2], you can call it in a `static` initializer or as part of another const context. This is why `static QUEUE: Queue<T,N> = Queue::new();` is possible (as long as `Queue::new` is const). Many embedded containers ensure their constructors are `const fn` for this reason. If a constructor couldn't be const (say it had to do complex setup), you'd fall back to the `StaticCell` pattern to do runtime init – but in our case, `Queue::new` **is** const, so you theoretically could fully initialize a static queue without even a `StaticCell`. The only reason we avoid `static mut Queue = Queue::new()` is the mutable access issue. But frameworks like RTIC circumvent that by controlling access at compile time (as shown above).

- **Compile-time checks:** Const generics also allow the compiler to check things like capacity bounds. For example, heapless might ensure `N > 1` at compile time. Some implementations require `N` to be a power of two (fring does this check via a trait bound or const assert). All of that happens at compile time, catching configuration errors early.

- **No unsafe needed for const eval:** In older Rust, getting a `'static` reference often involved unsafe code or macros, but with modern const generics and `const fn`, we can do a lot at compile time safely. The remaining trickiness (turning a `static` value into a mutable reference) is solved by patterns like `StaticCell` which we discussed.

In short, **const generics and const functions absolutely help define static queues safely**. They ensure our queue structure is truly static data, and the compiler can optimize and verify the code more easily. The design we are converging on – a `StaticCell` holding a `Queue<T,N>` – takes advantage of this: the `Queue` is const-initialized (so memory layout known, no runtime allocation), and `StaticCell` provides a one-time safe conversion to `'static mut`. This results in minimal runtime overhead: essentially O(1) operations for enqueue/dequeue (just index arithmetic and maybe an atomic flag check internally). In a no_std environment, you might need to wrap critical sections around enqueue/dequeue if using in interrupt handlers (to avoid concurrency issues if an interrupt and main code access simultaneously), but if it's truly one producer task and one consumer task (and e.g. interrupts use proper synchronization), the provided implementations handle memory ordering via atomic instructions (or disable interrupts in critical sections). For example, the heapless queue is implemented with such considerations (it even recommends using power-of-two capacities for performance) [1] [28].

Conclusion and Recommended Solution

To implement a zero-allocation, `unsafe`-free static SPSC queue for `lit-bit`, the research suggests the following **best approach**:

- **Use a proven ring buffer implementation** rather than writing one from scratch, to benefit from community testing and optimizations. The `heapless::spsc::Queue` is a strong candidate given it's already in use and meets the requirements. Alternatively, `BBQueue` or `fixed-queue` could be used if their APIs better fit (BBQueue for DMA scenarios, etc.), but for a generic actor mailbox `heapless::Queue` is simple and effective.

- **Allocate the queue statically**. In no_std, this means either declaring a `static QUEUE: Queue<T,N>` (with careful controlled access) or more flexibly, using `StaticCell`. The **safest pattern** is to have the user (or a macro) declare a `StaticCell<Queue<T,N>>` for each mailbox, and then initialize it during actor startup. This gives a `'static` queue without violating the no-unsafe rule. For example, lit-bit's new `create_mailbox_safe` does exactly this: it takes a `&'static StaticCell<Queue<T,N>>` and returns the `(Outbox, Inbox)` by calling `cell.init(Queue::new()).split()` [27]. This design should be continued. It might be slightly less ergonomic than a heap-based channel (since the user must provide the static storage), but it's the cost of no heap in embedded.

- **Ensure the capacity (`N`) and usage patterns are well-chosen**. Since we can't expand the queue at runtime, choose `N` large enough for worst-case actor message bursts. If backpressure or overflow is a concern, decide on a policy (drop oldest, drop new message, etc.). The current approach is fail-fast (return an error on full) [29], which is acceptable. Just document this clearly.

- **No `unsafe` in our code**: By using the above crates and patterns, we delegate any necessary `unsafe` to well-audited crates (heapless, etc.). Our code can remain marked `#!`

`[forbid(unsafe_code)]`. We should keep running tools like `cargo-geiger` to ensure no unsafe sneaks in. (As of the recent progress, `lit-bit-core` has 0 unsafe uses [30], which we want to maintain.)

- **Production readiness and formal assurance**: The approach described is already used in production embedded systems (e.g., RTIC examples, Ferrous Systems' projects, etc.), which gives confidence in its soundness. The algorithms are simple circular buffer operations that have been extensively tested. If further assurance is needed, one could consider using formal tools (e.g., model checking the enqueue/dequeue logic for race conditions), but given that our solution leans on established crates, we inherit their reliability. For instance, BBQueue's design was inspired by the **LMAX Disruptor** pattern (with known correctness in the field) [14], and heapless has been used in the Drone OS and other critical projects. There has also been academic work on verifying ring buffers and queue algorithms for real-time systems – while not specific to our crates, the consensus is that an SPSC ring buffer (with proper memory ordering) is one of the simpler lock-free structures to verify. In practice, sticking to these known patterns is far safer than crafting a novel approach from scratch.

- **Embedded (no_std) vs Std**: You may continue using separate implementations for std and no_std (as you do now via `cfg` features). That is fine, since on std you likely want an async channel to allow `.await` on send when full. But if you ever want to unify them, you could also consider using a *bounded sync channel* from `async-channel` or similar even in std (with a blocking send or poll), or use the same heapless approach in std for consistency (but then you lose the automatic async waiting – you'd need to poll or spin). A compromise could be to offer an async wrapper around the static queue (e.g., a future that waits for space, by checking the queue periodically or waking on receive). That, however, may complicate the no_std side. It might be simplest to keep using tokio's channel for std (which is proven and dynamic) and the static queue for no_std, abstracted behind the same `Inbox/Outbox` interface. This way, each platform uses the best tool available (async channel for std, static ring for no_std). The conditional compilation approach is working and is idiomatic for cross-platform Rust.

In conclusion, **implement a static SPSC queue by combining a const-generic ring buffer with a safe static initializer**. The community has converged on patterns like `heapless::Queue + StaticCell` as the go-to solution for exactly this problem – it yields a `'static` queue with zero heap, no unsafe, minimal overhead, and proven reliability. Adopting this in lit-bit (as has been started with `static_mailbox!`) aligns perfectly with the project goals of zero-cost abstractions and memory safety.

References:

- Heapless SPSC Queue (no_std, const-generic) – *"A statically allocated single producer single consumer queue with a capacity of N - 1 elements"* [1]. Provides `const fn new()` [2] for compile-time init.
- RTIC Framework Example – demonstrates splitting a static `heapless::Queue` into producer/consumer in an embedded context [4] (safe, no heap).
- `StaticCell` crate – *"reserve memory at compile time for a value, but initialize it at runtime, and get a 'static reference to it"* [5]. Example of initializing and using a `StaticCell` [6].
- Lit-bit safe mailbox init – uses `StaticCell` + `heapless::Queue` to create a `'static` mailbox without unsafe [27].

- BBQueue crate – SPSC queue based on BipBuffer (optimized for DMA), no_std and lock-free [11]. Static usage example with `BBBuffer::new()` and `try_split()` [31].
- Fixed-Queue crate – provides const-friendly, no_std, no_alloc containers (Vec, VecDeque, SPSC, MPMC) that are *lock-free and wait-free* [15]. Good for fully static systems.
- Fring crate – a fast no_std ring buffer for SPSC, emphasizing compile-time checked safety and lock-free operation [21] (requires power-of-two capacity).
- Drogue-Device actors – design note that system is "'static-centric" for actors and messages [23], reflecting the need for static mailboxes in embedded actor models.

---

[1] [2] [28] Queue in heapless::spsc - Rust
https://docs.rs/heapless/latest/heapless/spsc/struct.Queue.html

[3] [4] 'static super-powers - Real-Time Interrupt-driven Concurrency
https://rtic.rs/1/book/en/by-example/tips_static_lifetimes.html

[5] [6] [7] [10] static_cell - Rust
https://docs.rs/static_cell/latest/static_cell/

[8] [9] 2025-05-27.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/PROGRESS/2025-05-27.md

[11] [12] [13] [14] [31] GitHub - jamesmunns/bbqueue: A SPSC, lockless, no_std, thread safe, queue, based on BipBuffers
https://github.com/jamesmunns/bbqueue

[15] fixed-queue 0.4.7 - Docs.rs
https://docs.rs/crate/fixed-queue/0.4.7

[16] [17] [18] [19] [20] [21] fring - Rust
https://docs.rs/fring/latest/fring/

[22] Channel in embassy_sync::channel - Rust
https://docs.embassy.dev/embassy-sync/git/default/channel/struct.Channel.html

[23] [24] Introducing Drogue Device — Drogue IoT
https://blog.drogue.io/introducing-drogue-device/

[25] [26] [29] [30] 2025-05-25.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/PROGRESS/2025-05-25.md

[27] mod.rs
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/lit-bit-core/src/actor/mod.rs