ChatGPT

Embassy Channel Error Handling in 0.6

`.send()` Behavior and Failure Possibility

Infallible API: In Embassy 0.6, the asynchronous `.send()` on an `embassy_sync::channel::Sender` returns a `SendFuture` that yields `()` on success, with no `Result` type. There is no built-in error for a failed send – the API is designed to be infallible under normal usage [1] [2]. In practice, `.send().await` will wait until there is space in the channel's buffer and then enqueue the message, never returning an error. It does **not** signal failure if a receiver is missing. The only time a panic could occur is due to misuse (e.g. polling a completed future twice, which triggers an internal assert in `SendFuture` [3]), not because of channel state.

Dropped receiver scenario: If the corresponding `Receiver` task is dropped (i.e. no consumer is actually waiting on the channel), the channel **does not close or error**. Instead, the send operation will still enqueue the message if the buffer isn't full, or block indefinitely if the buffer is full (since no receiver will ever make space) [4] [5]. There are no runtime checks that "unlock" the sender in this case – the channel remains open and will simply never be drained. In effect, dropping all receivers can lead to a *deadlock* where senders wait forever once the buffer fills up. This is analogous to the behavior of older channel implementations like Crossbeam's MPMC channel before they added a closure mechanism – the senders would hang indefinitely if the receiver went away [6] [7]. In short, **Embassy channels cannot be closed** in 0.6, so `.send().await` will neither return an error nor panic due to a closed channel; it will just stall if the message can't be delivered (potentially forever, if no receiver ever consumes it).

Embassy's Channel Lifecycle and Design Philosophy

Always-open channels: The design of Embassy's channels assumes a static or long-lived lifecycle. There is no concept of an explicit "channel closure" or termination signal when receivers are dropped. In embedded `no_std` contexts, tasks (and their associated channels) are often initialized once and run for the lifetime of the system (e.g. many Embassy examples spawn tasks that loop forever). Indeed, frameworks built on Embassy like Ector define actor tasks that never return – for example, an actor's `on_mount` handler runs an infinite loop processing messages (`-> !` never type) [8]. This pattern implies the channel backing an actor's inbox is meant to live as long as the actor task (ideally for the program's duration), and not be shut down arbitrarily.

No internal disconnect tracking: Embassy channels do not count receivers or flag a disconnected state. The channel implementation lacks a `Disconnected` or `Closed` variant for send errors, and there's no internal refcount to determine if "all receivers have gone away." The **absence of a `Closed` error** is evident in the `TrySendError` type – it only has a `Full` variant and nothing like Tokio's `Disconnected` [9] [10]. This reflects Embassy's philosophy: channels are meant to be always available and generally not dropped at runtime. The user is responsible for ensuring a receiver is listening, or otherwise the system will hit a deadlock or drop messages silently. In practice, an Embassy channel is often a static buffer tied to static tasks, so the notion of closing is mostly irrelevant in the intended use cases.

Long-lived usage: Because of this design, it's expected that an Embassy `Channel` lives until either the program ends or perhaps until a full system reset. If an actor or task needs to end, typically you would coordinate that logic at a higher level (for instance, by stopping producers or using a flag) rather than by closing the channel. There is a `Channel::clear()` method that an application *could* call to empty the buffer and wake any blocked senders [11] [12], but this is a manual intervention – not an automatic "drop" behavior. In summary, Embassy's channels favor simplicity and predictability in always-on scenarios: no dynamic teardown, no hidden errors – but also no automatic relief if you misuse them by dropping receivers.

Behavior of `.try_send()` and Error Semantics

`try_send` **only signals fullness:** The non-waiting variant `.try_send()` returns a `Result<(), TrySendError<T>>`. In Embassy 0.6, `TrySendError` **only has one variant**, `Full(T)` [9], which indicates the channel's buffer was at capacity and the message couldn't be enqueued. There is **no** `TrySendError::Closed` variant or similar. This means that `try_send()` will never report a "closed channel" error – if you call `try_send()` and the channel has any free capacity, it will succeed (even if no receiver is actively processing messages). If the buffer is full, you get a `Full` error with your message returned, and it's up to you to retry or drop that message.

No "Closed" condition: Because the channel isn't considered closed when receivers vanish, `TrySendError::Closed` simply doesn't exist in Embassy's API (unlike `std::mpsc` or Tokio which have a Disconnected/Closed error). Practically, if all receivers are gone, `.try_send()` will continue to behave as follows: until the buffer fills up, it will return `Ok(())` (placing messages into the queue that nothing will ever receive). Once the buffer is full, further `try_send()` calls will return `Err(TrySendError::Full(_))` – not some "Closed" error – because from the channel's perspective it's just perpetually full. There's no built-in way for the sender to distinguish "full because slow consumer" from "full because no consumer at all." In other words, `TrySendError::Closed` **never occurs** in Embassy 0.6 channels, by design. The only error state is a full queue [9].

Backpressure vs. silent drop: One important consequence is that if a receiver task stops polling the channel (or is dropped), senders will experience backpressure as if the channel were just busy/full, rather than an immediate error. Some users in the embedded Rust community handle this by using `try_send()` in a loop or with a strategy to drop messages when `Full` occurs (to avoid blocking the sender task). For example, one community recommendation for high-rate producers is to call `try_send()` and simply drop the message on `TrySendError::Full` instead of awaiting indefinitely [13]. This at least prevents a lockup by shedding load, but it's an application-level policy. The key point is that **a "closed" channel is not signaled** – you either block (with `.send()`) or get a `Full` error (with `.try_send()`), but never a disconnect notification.

Error Handling Practices in Embassy-Based Actor Systems

Implicit failure handling: Because Embassy channels don't error on disconnect, many Embassy-based actor frameworks simply assume infallible message delivery (so long as the actor is running). For instance, the Ector actor framework (which integrates with Embassy) does not propagate send failures – its actor `Address.notify()` returns an `Awaitable` that doesn't yield a `Result` in the failure-free case. In Ector's design, actors are spawned with a fixed-size inbox (Embassy `Channel`), and the actor's task runs

forever, reading from that inbox [8]. There's no code path for an actor mailbox closing while the system is running; as a result, there's no need to handle "address not alive" errors in normal operation. This mirrors the typical embedded pattern: tasks (and their message channels) are not dynamically killed off; they often only terminate at system shutdown or reboot.

Actor shutdown considerations: If an Embassy-based actor *does* need to shut down or be restarted (which is somewhat unusual), it becomes the developer's responsibility to handle that gracefully. Since dropping a receiver leaves senders in limbo, one must implement a protocol for shutdown. Common approaches might include: sending a special "stop" message that makes the actor exit its loop, then **not sending any further messages** to that actor's channel; or using a `Signal`/`Notification` primitive to coordinate task termination separately. After an actor stops, its channel is typically just left idle (or you might call `clear()` to drop pending messages and wake any blocked senders as a form of manual cleanup [14]). But notably, frameworks and examples do not show a built-in mechanism for "closing" an actor's inbox – they avoid the situation entirely by design. Real-world Embassy applications tend to either never drop the receiving task, or perform a full system reset when such a major change is needed.

Ignoring delivery failures: Given the above, it's common (if a bit unsettling) that Embassy actor systems simply **ignore the possibility of message delivery failure**. There is an implicit contract that as long as you keep the actor alive, any `.await` on `send()` will eventually succeed. If that contract is broken (e.g. the actor died), the system doesn't provide an error; the problem manifests as a stuck `.await` or a filled queue. In practice, developers mitigate this by careful design: ensuring each channel has a permanently running consumer, sizing channel capacities appropriately, and sometimes monitoring system health via watchdogs. Some projects also use timeouts or watchdog tasks to detect if a certain message response never arrived, as an indirect way to catch a stuck actor. But *explicit* error propagation for lost messages is generally not provided by Embassy's channel abstraction itself.

Community guidance: The lack of an error on disconnect is a deliberate trade-off for simplicity, but it's noted in community discussions as a potential footgun. For example, the Crossbeam channel had a similar design and later introduced channel closure on receiver drop to prevent deadlocks and memory leaks [15] [16]. In the embedded realm, developers are often aware that dropping a task (and its channel) is dangerous unless you also stop senders. The **community recommendation is usually to never drop all receivers without also stopping senders**, since otherwise senders will either block forever or keep queuing into / filling the buffer silently. Actor frameworks built on Embassy therefore typically don't even expose a concept of actor "shutdown" at runtime – avoiding the issue altogether.

Should Embassy `.send()` be Wrapped in a `Result`?

Consistency vs. reality: In a multi-runtime actor system like `lit-bit-core`, it's understandable to want a uniform interface – e.g., always returning a `Result` from `Address.send()` – to mirror Tokio's fallible send. Wrapping Embassy's infallible send in a `Result` (perhaps with a dummy error type or an Option) could be seen as defensive design or future-proofing. However, currently this would be a **semantic no-op**: there is no error to catch unless you implement extra logic outside of Embassy's channel. In Embassy 0.6, a send failure can only happen if you deliberately simulate one (for instance, by checking some external flag that the actor has stopped).

Silent failure concerns: From an architectural correctness standpoint, relying on silent message loss or hang is not ideal. If there is any chance an actor's receiver might not be listening, explicit error handling is preferable to failing quietly. Thus, adding a `Result` to Embassy sends could help flag logically unexpected situations (e.g., "why did this actor not get my message?"). That said, without modifiying the channel itself, an Embassy-based `Address.send()` would always return `Ok(())` in practice. Some developers might choose to implement a custom error mechanism – for example, an atomic "alive" flag that the actor clears on exit, which the `Address` checks before sending. This would allow the `Address` to return an error if the actor is known to have stopped. Implementing such checks incurs extra complexity and overhead, and is not part of Embassy's out-of-the-box design.

Pragmatic approach: Many embedded frameworks choose the simpler route: assume infallibility and document the behavior. For instance, one might document that **sending to a stopped actor is undefined behavior** (or will just block forever) in the Embassy backend, and encourage avoiding that scenario. The Tokio backend can still return an error (because Tokio channels do support closure), but on Embassy the same high-level `Address.send()` method could either return a dummy `Result<(), Never>` (an Ok that never is Err), or perhaps a `Result<(), ClosedError>` where `ClosedError` never actually happens unless you manually trigger it. Either approach leads to asymmetry in practice, which is exactly the inconsistency the `lit-bit-core` author is concerned about.

Recommendation: If consistency and future-proofing are priorities, **wrapping Embassy's send in a** `Result` (even if it's always `Ok` in the current implementation) can be justified. It would make it easier to handle both runtimes uniformly in user code and signal the *theoretical* possibility of failure. This could guard against future Embassy updates that might introduce channel closure semantics, or against logic bugs where an actor might not be receiving. On the other hand, if performance and simplicity are paramount, one might decide that on Embassy targets the send can be considered infallible and thus omit the `Result` for that backend. In either case, it's critical to communicate the behavior: **silent message loss is possible** if an Embassy actor task stops unexpectedly, since the channel won't report any error. In systems where such a scenario is unacceptable, you should enforce at a higher level that actors don't drop receivers without coordinating with senders (or use an alternate mechanism like a oneshot notification to signal task completion).

In summary: Embassy's channel API chooses silent blocking over explicit failure. It assumes a design where channels live indefinitely and receivers are always present, which is typical in embedded applications. If your architecture might violate those assumptions, you'll need to add your own error-handling layer or design patterns (like timeouts or heartbeat checks) to detect and handle lost consumers. Otherwise, embracing Embassy's infallible send – with the understanding that it won't fail unless something has gone seriously wrong – is a reasonable approach. Just be cautious: a "never-failing" send in Embassy is only as safe as the guarantee that your receiver is alive and well. If that guarantee might not hold, then introducing a `Result` in your API (and the checks to back it) is a wise defensive measure, even if it feels redundant in the current Embassy 0.6 model.

Sources:

- Embassy channel documentation (0.6/0.7), showing bounded MPMC behavior with backpressure and no mention of closure [17] [9].
- Embassy channel source code, confirming `Sender::send()` awaits until buffer space is free and never returns an error [4] [5].

- Definition of `TrySendError` in Embassy – only contains a `Full` variant (no `Closed`) [9].
- Crossbeam channel issue illustrating the deadlock/memory-leak problem when a channel has no receivers (similar outcome as Embassy's always-open channel) [6].
- **Ector** actor framework example, where actor tasks run indefinitely (`-> !`), reflecting the expectation that channels are not dropped mid-execution [8].
- Community commentary on using `try_send()` and dropping messages on overflow as a strategy to avoid blocking in embedded contexts [13].

---

[1] [2] [3] [4] [5] [9] [10] [11] [12] [14] [17] channel.rs - source
https://docs.rs/embassy-sync/latest/src/embassy_sync/channel.rs.html

[6] [7] [15] Potential deadlock and resource leak when `Receiver` is dropped in `crossbeam_channel`'s bounded channel · Issue #1102 · crossbeam-rs/crossbeam · GitHub
https://github.com/crossbeam-rs/crossbeam/issues/1102

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

[13] muji_tmpfs (u/muji_tmpfs) - Reddit
https://www.reddit.com/user/muji_tmpfs/comments/

[16] crossbeam-rs/crossbeam issues and pull requests | Ecosyste.ms ...
https://issues.ecosyste.ms/hosts/GitHub/repositories/crossbeam-rs%2Fcrossbeam/issues