ChatGPT

Integrating Static Tasks in an Embassy 0.6 Async Actor System

Overview: This report discusses how to implement a zero-heap async actor system on embedded platforms using Embassy 0.6, focusing on static task allocation and Embassy integration. We cover best practices for spawning tasks with `StaticCell`, choosing an appropriate mailbox mechanism (Channel vs Pipe), using the Embassy `Spawner`, organizing static memory in real projects, handling timers in actor tasks, and ensuring graceful error handling. We also compare Embassy's channels with Tokio's in terms of memory and performance. The guidance assumes a GAT-based actor trait (generic over message type with async functions) and aims to match a Tokio-based implementation's API while respecting embedded no-heap constraints.

Static Task Spawning with `StaticCell` (Embassy 0.6)

Static Allocation Requirement: Embassy tasks must have a `'static` lifetime. Unlike Tokio (which can allocate tasks on the heap), Embassy targets no-std environments and cannot spawn futures with non-static lifetimes. In practice, **any data or actor instance used inside an Embassy task must be stored in static memory**. This is typically done with the `static_cell` crate (or Embassy's re-export of it) to allocate memory at compile-time and initialize it at runtime [1] [2]. The `StaticCell<T>` type provides a safe way to get a `&'static mut T` reference by calling `.init()` once.

Best Practice – Using `StaticCell` to Spawn an Actor Task: Allocate the actor (and any large data structures it owns) in a `StaticCell` before spawning its task. For example, one might write:

```
static ACTOR_CELL: StaticCell<MyActor> = StaticCell::new();

let actor: &'static mut MyActor = ACTOR_CELL.init(MyActor::new(...));
// Now `actor` is a `&'static mut` reference, which can be moved into an Embassy
task.
unwrap!(spawner.spawn(run_actor(actor)));
```

Here, `run_actor()` would be an `async fn` that takes a `&'static mut MyActor` and runs the actor's message loop. By obtaining a static mutable reference, we satisfy the `'static` requirement of Embassy tasks [2] [3].

Ownership and Lifetime Handling: When passing the actor into the task, you have two common patterns:

- *By Mutable Reference:* As shown above, give the task a `&'static mut MyActor`. The task can then treat the actor as a pinned-in-memory singleton. Ensure that only one task uses that mutable

reference (to avoid aliasing). The `StaticCell` guarantees the value is initialized exactly once and then yields a unique `&'static mut` pointer [1] [4].

- *By Value (Move):* Alternatively, move the actor into the task future. Since the actor is allocated in static memory, you could do `let actor_ref = ACTOR_CELL.init(actor_instance); spawner.spawn(async move { actor_main(actor_ref).await });`. The actor reference is `'static`, so moving it is fine. This approach is effectively similar to passing the static reference as an argument.

In both cases, the key is that the actor lives forever (or until the device is reset). **Embassy tasks cannot take owned parameters with limited lifetimes** – any non-`'static` lifetimes or generic parameters will cause compile errors [2]. One user noted that "Embassy tasks can't be generic (including lifetimes), you'll need to pass `LEDC<'static>` to it in some form" when encountering a peripheral with lifetime [2]. The solution was to allocate that peripheral in a `StaticCell` so it could be passed as `&'static` into the task [5] [6].

Example – Spawning with StaticCell Macro: Many projects define a helper macro for concisely creating static allocations. For instance, the pattern below reserves static space and initializes a buffer:

```
macro_rules! singleton {
    ($val:expr, $T:ty) => {{
        static STATIC_CELL: StaticCell<$T> = StaticCell::new();
        STATIC_CELL.init($val)
    }};
}

// Usage:
let buffer: &mut [u8; 240] = singleton!([0_u8; 240], [u8; 240]);
```

This ensures `buffer` is a `'static` memory block [7] [8]. You can similarly allocate static actors or other structures. The [ector] crate (an async no-alloc actor framework) uses a procedural macro to automate this pattern. For example, `ector::spawn_actor!(spawner, NAME, Type, init_value)` expands to create a static `StaticCell` for the actor and spawn its task, returning a handle (address) for messaging [9] [10]. This macro encapsulates the StaticCell logic, ensuring each actor has a dedicated static allocation.

TIP: Use `StaticCell::init_with(|| ...)` when the initialization depends on other runtime values. This variant takes a closure to construct the object in-place, avoiding intermediate stack copies [11]. In the ESP32 LEDC example, the user did:

```
let ledc = LEDC.init_with(|| LEDC::new(peripherals.LEDC, clocks, &mut
system.peripheral_clock_control));
```

This constructed the `LEDC` driver inside the static cell at runtime [12]. Using `init_with` is especially useful when the value's constructor needs references to other static data (like `clocks` above). It ensures everything is properly initialized in static memory with minimal overhead.

Summary: *Allocate each actor (and other long-lived objects) in a* `StaticCell` *to get a stable* `'static` *reference. Pass those static references into* `Spawner.spawn(...)` *so the tasks own them for their lifetime. This avoids dynamic memory and satisfies Embassy's requirement for* `'static` *futures* [3]*. Properly manage uniqueness of mutable references – once you* `init()` *a StaticCell, use that data in only one task unless you wrap it in an* `Mutex` *or other sync primitive.* (In an actor system, typically one task = one actor state, so this is naturally one-to-one.) Finally, note that **calling** `.init()` **more than once on the same StaticCell will panic** [13], so ensure each StaticCell is initialized exactly once.

Actor Mailboxes: Choosing `Channel` vs `Pipe`

Each actor needs a mailbox (message queue) to receive messages asynchronously. Embassy provides two primary queue abstractions in `embassy_sync`:

- **Channel:** a generic multi-producer, multi-consumer queue for any type `T`.
- **Pipe:** a byte-oriented ring buffer (for `u8` only).

For an actor mailbox (where messages are Rust structs or enums), **Embassy's MPMC** `Channel` **is the idiomatic choice**. It allows tasks to send typed messages to the actor task with backpressure if the buffer is full [14] [15]. In contrast, `Pipe` is designed for byte streams (like UART data or file I/O streams) and only buffers raw bytes [16] [17]. Using a Pipe for general actor messages would require manually serializing/deserializing message types into bytes, adding complexity and overhead.

Channel Basics: An Embassy `Channel` is declared as a `static` with a given capacity and a mutex type. For example:

```
use embassy_sync::channel::Channel;
use embassy_sync::blocking_mutex::raw::ThreadModeRawMutex;

static MAILBOX: Channel<ThreadModeRawMutex, MyMessage, 8> = Channel::new();
```

This creates a bounded channel for `MyMessage` with capacity 8. The mutex type (`ThreadModeRawMutex` here) is chosen based on context: - `ThreadModeRawMutex` is used when all senders/receivers run in thread mode on a single core (no interrupts), offering a very low-cost lock (essentially no locking, since Embassy's thread-mode executor is non-preemptive) [18]. - If messages can be sent from interrupts or other contexts, a `CriticalSectionRawMutex` is often used to disable interrupts during queue operations [19]. This prevents data races at the cost of a brief critical section.

To use the channel, you obtain a sender and receiver:

```
let sender = MAILBOX.sender();
let receiver = MAILBOX.receiver();
```

The sender can be cloned to give to multiple producer tasks (it implements `Clone` / `Copy` for `ThreadModeRawMutex` usage). The receiver is typically held by the actor task to `.receive().await` messages in a loop. **Each message sent is delivered to exactly one receiver** (in this case, our single actor inbox) [20].

Why Channel is Preferred for Actor Mailboxes:

- **Type Safety & Ergonomics:** Channel directly handles your `Message` type. You `sender.send(msg).await` and `receiver.receive().await` to get a `MyMessage` value [21]. This is straightforward and avoids manual buffer management. In real-world code, this leads to clear actor loops like:

```
loop {
    let msg = receiver.receive().await;
    self.handle_message(msg).await;
}
```

Compare that to using a Pipe: you would need to define a byte protocol or length-prefix, then write bytes and read bytes, and reconstruct messages. That's unnecessary overhead for in-process actors.

- **Memory Usage:** The `Channel<Mutex, T, N>` allocates a fixed array of `N` messages of type `T` internally. This static size means **memory use is predictable** and set at compile-time [22]. For example, a `Channel<..., MyMessage, 32>` will take space for 32 instances of `MyMessage` plus some bookkeeping. A `Pipe<M, N>` allocates N bytes buffer, which might seem smaller if `MyMessage` is large. However, to use a Pipe for arbitrary messages you'd likely allocate a similar total size for bytes (e.g. if messages are up to 16 bytes, you'd need N*16 bytes buffer for N messages). Any advantage is lost unless messages are raw byte streams to begin with (which actor messages usually are not).

- **Latency and Throughput:** Embassy's channels provide asynchronous send/receive with backpressure. Under the hood they use atomic operations or critical sections to enqueue/dequeue, which on a microcontroller is very fast (a few CPU instructions plus potential interrupt masking). The **latency to send a message** is on the order of microseconds, and receiving is similar – essentially the cost of checking buffer pointers and possibly waking the task waiting on the queue. `Pipe` has comparable complexity for each byte transferred. For multi-byte messages, `Pipe` would incur multiple writes/reads (one per byte) unless you batch them, whereas `Channel` enqueues the whole struct at once. Thus, for typical actor message patterns (discrete messages), Channel is more efficient and simpler.

- **Real-World Usage:** The community consistently uses `Channel` for task communication in Embassy. For instance, Eloy Coto's embedded tips blog demonstrates a static channel for message passing between tasks, recommending Embassy channels because "they provide a nice abstraction...

without problems and race conditions" and noting that *"the computational overhead isn't that high"* [23] [24]. Ector, the embedded actor framework, internally uses channels (or similar constructs) for inboxes (exposed via an `Inbox` trait). It doesn't use pipes for message delivery – messages remain typed.

When Might `Pipe` Be Used? Pipes shine when streaming bytes from one task to another, such as feeding a logging task or handling a continuous data stream like sensor readings or file data. They integrate with Rust's `AsyncRead/AsyncWrite` traits (the `Reader` and `Writer` from a `Pipe` implement these) [25]. If your actors were dealing with raw byte streams (e.g. one actor pumps a TCP socket and another processes bytes), a Pipe could be appropriate between those two tasks. But for an **actor message mailbox (discrete events/commands)**, a Channel is far more straightforward.

Memory Footprint Comparison: In a scenario where messages are 8 bytes long and you want to buffer up to 10 messages: - A `Channel<Mutex, [u8; 8], 10>` will statically allocate 10*8 = 80 bytes for storage, plus a few bytes for head/tail indices and wakers. - A `Pipe<Mutex, 80>` could similarly buffer 80 bytes, but then you must pack/unpack messages into the byte stream. The overall RAM usage is similar, but the `Pipe` requires additional code to manage message boundaries.

Additionally, Channel can buffer complex types (including enums or structs) in place without conversion. This avoids any serialization overhead (CPU time and code size) that a Pipe-based scheme would impose.

Latency Comparison: Both Channel and Pipe operations are O(1) for sending or receiving a single element (amortized). Channel's operations involve a single enqueue or dequeue of a struct; Pipe's involve writing/reading potentially multiple bytes. In practice, the difference is negligible for small messages on microcontrollers. Embassy's executor will suspend the sending task if the Channel is full, or the receiving task if the Channel is empty, and resume them efficiently when data is available [15]. The **end-to-end message latency** (time from send to receive when the actor is waiting) is dominated by how quickly the actor task gets scheduled, which is on the order of microseconds to a millisecond (depending on system load and clock speeds). There's no significant added latency from using Channel vs Pipe.

Conclusion: *Use* `embassy_sync::channel::Channel` *for actor mailboxes.* It is purpose-built for async message passing with strong typing and fixed capacity. `embassy_sync::pipe::Pipe` is not suited for general actor messages – it's a specialized tool for byte streams (similar to a FIFO buffer for bytes) [26]. By using Channel, your actor code remains clean and your memory usage is clearly defined at compile time. For example, a bounded channel of 32 messages in a critical-section mutex is a common, robust choice for inter-task messaging [27].

Example: A simple actor setup might be:

```
static MAILBOX: Channel<CriticalSectionRawMutex, ActorMsg, 16> = Channel::new();
// ...
#[embassy_executor::task]
async fn actor_task(mut actor: MyActor, mut rx:
embassy_sync::channel::Receiver<'_, CriticalSectionRawMutex, ActorMsg, 16>) {
    loop {
        if let Some(msg) = rx.receive().await {
```

```
            actor.handle(msg).await;
        }
        // (If receive() returns None, it could indicate all senders dropped –
handle termination if needed)
    }
}
```

This pattern is used in many embedded projects (for instance, in Drone OS or Drogue Device actors, etc.) to safely pass commands to dedicated tasks.

Using the `Spawner` in Embedded Contexts

In an Embassy application, the `Spawner` is a handle used to spawn tasks on the executor. It is provided to your `#[embassy_executor::main]` function and any tasks you spawn during initialization. **The recommended practice is to pass the `Spawner` to any code that needs to spawn new tasks**, rather than making it a global variable.

Spawner Characteristics: - `Spawner` is a lightweight, `Copy` type [28]. Copying it does not clone an underlying heap or anything – it's essentially a reference to the executor's task queue. - The `Spawner` can only be used from the executor's thread (i.e., not from interrupts unless you've set up an executor on that core/thread). Typically, you use it within async tasks or `main` before initialization completes.

Access Patterns:

- **Pass Explicitly to Tasks:** When you initially run the executor with `executor.run(|spawner| { ... })`, you get a `Spawner`. You can immediately use it to spawn top-level tasks. If some of those tasks need to spawn further tasks, you should give them a copy of the Spawner. For example, you might spawn a supervisor task as `spawner.spawn(supervisor_task(spawner))…` so that this supervisor can use `spawner` internally to launch other tasks (the Spawner being `Copy` makes this ergonomic) [29] [28]. This approach is explicitly suggested in Embassy's docs: *"To spawn more tasks later, keep copies of the Spawner (it is Copy), for example by passing it as an argument to the initial tasks."* [28]. Each task that needs to spawn others can thus receive a `Spawner` at creation.

- **Store in a Static (Global) if Necessary:** In some designs, you might want a globally accessible spawner (for convenience in library code, etc.). This isn't generally needed, because you can structure your application to propagate the spawner where required. But if absolutely necessary, one could store a `Spawner` in a `static mut` or a `StaticCell` and make it accessible globally. **This is not the preferred method**, as it introduces unsafety (if using `static mut`) and can make reasoning about task creation harder. The Embassy executor requires `&'static mut Executor` anyway (for initialization), which is often done via StaticCell [30]. So you already have a global executor; the spawner is just a handle to it. It's better to pass that handle around than to hide it in a global.

- **Using `Spawner` within Actors:** If your actor needs to spawn child tasks (for example, to offload some work or create a helper actor), you have a couple of options. You can give the actor's constructor or `.start()` method a Spawner to hold onto. Alternatively, design the actor's API such

that any spawning is initiated from outside (e.g., the supervisor or main task spawns all actors and passes them their dependencies). Many embedded applications statically spawn all tasks at startup (since the set of tasks is often fixed), in which case dynamic spawning from within an actor is not common. But if you do dynamic spawning, treat `Spawner` like a capability that you pass in as needed.

Example: In the ector actor framework, the `spawn_actor!` macro requires the spawner as an argument, precisely following this paradigm [9]. It spawns the actor's task immediately using the given spawner. Similarly, the Embassy executor's examples often show code like:

```
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    spawner.spawn(task1()).unwrap();
    spawner.spawn(task2(spawner)).unwrap(); // passing spawner to task2
}
```

Here, `task2` could use the spawner to spawn more tasks (or spawn sub-tasks on events). This explicit passing makes the control flow clear.

Why not a Global Spawner? Aside from style, there's a practical reason: The `Spawner` is only valid after the executor is running. If you stash it in a global before that, you must ensure not to use it too early. Also, since it's `Copy`, accidental misuse is possible if global. By passing it through function arguments, the compiler helps ensure it's used in the right places (during async execution). The Embassy docs warn that `Executor.run()` requires `&'static mut Executor` and suggest `StaticCell` for storing the Executor [30] – once running, the provided `Spawner` should be treated similarly carefully.

Spawner and Static Tasks: You might wonder if you can avoid `Spawner` entirely by declaring all tasks with the `#[embassy_executor::task]` attribute and letting the executor auto-spawn them. Embassy's attribute macro indeed allocates tasks statically and can start one main task. However, for any additional tasks, you still need to spawn them (usually in the `main` task using the Spawner). The only "implicit" spawning done by the macro is launching the main task. Everything else is under your control via `Spawner`.

Summary: *Use the* `Spawner` *by passing it where needed, rather than global state.* This explicit approach is recommended in the Embassy docs [28]. It avoids unsafe global mutable state and makes the task structure clearer. Since `Spawner` is `Copy`, distributing it is trivial and has no performance cost. For instance, an initialization routine can spawn multiple subsystem tasks and also hand them the spawner if those subsystems may create further tasks. In truly static scenarios (fixed task set), you may not need to propagate the spawner beyond `main` at all – spawn all tasks up front.

Memory Layout and Static Allocation Patterns in Real Projects

Real-world embedded Rust projects using Embassy demonstrate a variety of patterns for organizing statically allocated memory. The common theme is **all long-lived resources (executors, drivers,**

peripherals, buffers, actors) are allocated in `static` storage, either via `StaticCell` or plain `static mut` (with care). Here are some patterns and examples:

- **Grouping into a Single Static Struct:** Some projects define a composite struct that holds all major resources, then allocate one instance of that struct in a `StaticCell`. For example, an ESP32 project defined:

```
pub struct SystemInit<'a> {
    executor: Executor,
    clocks: Clocks<'a>,
    ledc: LEDC<'a>,
    hstimer0: Timer<'a, HighSpeed>,
}
static SYSTEM_INIT: StaticCell<SystemInit> = StaticCell::new();
```

Then in `main`, they initialized this struct in one go (or in pieces) inside the static cell [31] [32]. This approach can simplify initialization order and ensure all related components live together. However, it may require `unsafe` if you need to get `'static` lifetimes internally (here they ended up not using the single struct, but it's a considered approach).

- **Multiple StaticCells for Components:** It's very common to see multiple static items, one per component. For instance, one project had:

```
static EXECUTOR: StaticCell<Executor> = StaticCell::new();
static CLOCKS: StaticCell<Clocks> = StaticCell::new();
static LEDC: StaticCell<LEDC> = StaticCell::new();
static HSTIMER0: StaticCell<Timer<HighSpeed>> = StaticCell::new();
```

and so on [33]. Each is initialized individually in `main()` (as shown earlier in the LEDC example) [12] [34]. The benefit is clarity – each static is named and used for one purpose – and you can initialize them in the right order with `.init` and `.init_with` calls. The downside is a bit of boilerplate and the need to ensure consistency (e.g., an array length constant might need to match between definition and usage, as seen in an `esp-embassy-config` example for config entries [35] [36]).

- **Using Macros for Static Allocation:** As mentioned, defining a `singleton!` macro is popular. The example in the MIPI DSI (display) driver issue shows how they allocate buffers and driver objects with one macro call [7] [8]. This hides the `StaticCell` mechanics and makes the code terser. Some HAL crates or board support crates might provide such macros for common resources.

- **Pools or Arrays of Task Storage:** If an application needs to spawn a variable number of identical tasks (e.g., a pool of worker tasks), you can leverage Embassy's ability to reuse finished tasks. The `#[embassy_executor::task(pool_size = N)]` attribute can allocate `N` task slots for a given task function. Alternatively, manually, you could create an array of `StaticCell<TaskStorage<...>>` if using the low-level API [37] [38]. In practice, most embedded

systems don't dynamically create many tasks at runtime; they start with a fixed set. Thus, static definitions per task are typical.

- **Example Repo – Networking Stack (rp-pico WiFi):** In an example setting up a Wi-Fi stack on a Raspberry Pi Pico W (RP2040 + CYW43 WiFi chip), the author used static cells for the network stack and resources:

```
static STACK: StaticCell<Stack<WifiDriver>> = StaticCell::new();
static RESOURCES: StaticCell<StackResources<4>> = StaticCell::new();
```

They then did `let res = RESOURCES.init(StackResources::<4>::new());` and `let stack = STACK.init(Stack::new(driver, config, res, etc.))`. The network task is spawned with these static references [39]. This shows how even complex systems (TCP/IP stacks, etc.) are fit into static memory with known sizes (here a pool of 4 network packets in `StackResources<4>`).

- **Actors/Devices as Static Singletons:** Often each hardware device (or actor) is represented as a singleton task with a static channel and static state. For example, imagine an embedded app with a sensor actor and a display actor. You might see:

```
static SENSOR_ACTOR: StaticCell<SensorActor> = StaticCell::new();
static SENSOR_MAILBOX: Channel<..., SensorMsg, 4> = Channel::new();
static DISPLAY_ACTOR: StaticCell<DisplayActor> = StaticCell::new();
static DISPLAY_MAILBOX: Channel<..., DisplayMsg, 8> = Channel::new();
```

In `main`, you initialize the actors (`let sensor = SENSOR_ACTOR.init(SensorActor::new());`) and then spawn their tasks, passing each its mailbox receiver. This pattern is essentially what ector's `spawn_actor!` macro does behind the scenes [9] – it defines a `static` for the actor and a static channel for its inbox (with capacity often defined via a const or default).

- **Memory-Constrained Devices:** On very RAM-limited devices, developers carefully size each static buffer. For instance, one might see a global byte buffer for DMA or for a graphics framebuffer allocated with `StaticCell<[u8; SIZE]>`. By keeping these as `static` with clear sizes, it's easy to see the total RAM usage. Embassy doesn't use the heap at runtime, so all memory use comes from these static allocations plus the stack. The Embassy book notes that *"the executor doesn't have fixed-capacity data structures"* in the sense that tasks are allocated in an arena and if you spawn beyond the arena it panics [40] [41]. In practice, that "arena" is also backed by static memory (the task storage), so it's deterministic. Projects often define the total task arena size via features or just trust the sum of tasks fits.

Gotchas and Patterns:

- **Ensure Alignment and Initialization Order:** If using multiple static cells that depend on each other (like a peripheral clock and a driver that uses that clock), be mindful of initialization order in `main()`. Using `init_with` closures can capture already-initialized static refs in later static inits

[12] [34]. This ensures, e.g., the `Clocks` are initialized before you construct a driver that needs `&'static Clocks`. Always call `.init` on dependencies first, then on dependents.

- **Static Mut vs StaticCell:** You *can* use raw `static mut` and `MaybeUninit` to allocate static memory, but this is unsafe. `StaticCell` internally uses atomics/critical-section to ensure you only init once and get a safe reference [42] [43]. Real projects overwhelmingly favor `StaticCell` for safety. Only in bootloaders or extremely low-level code might you see `static mut BUF: MaybeUninit<T> = MaybeUninit::uninit();` and then later an `unsafe { BUF.as_mut_ptr().write(val) }`. Stick with `StaticCell` unless you have a compelling reason otherwise.

- **Example Repos:** You may find it useful to browse projects like [drogue-device] (which uses ector and Embassy) or embedded firmware examples in the Embassy repository (the `examples/` directory for various boards). For instance, the Embassy `blinky_two_channels.rs` example shows two tasks toggling an LED via a static Channel [44] (with `static CHANNEL: Channel<ThreadModeRawMutex, LedState, 64> = Channel::new();` etc.) [45]. Also, the `esp-embassy-wifihelper` crate example shows static channels for WiFi events [46]. Studying these will reinforce how static data is laid out.

In summary, **real-world embedded projects using Embassy allocate virtually everything statically**. Common patterns include one static cell per major component (executor, peripherals, actors), sometimes grouped in structs, and liberal use of `Channel` for any inter-task communication. This results in a memory layout that is fixed and knowable, critical for embedded reliability. By following these patterns – many of which are demonstrated in open-source projects – you ensure your actor system has a solid, static foundation.

Timers in Async Actors: Cancellable and Composable Patterns

Working with timers in an async context is a common need (for timeouts, delays, periodic tasks, etc.). Embassy provides `embassy_time::Timer` which you can `.after(duration)` to get a future that completes after a given time. Integrating `Timer::after` into actor logic in a **cancellable or composable** way typically involves using `select`*-like patterns* or spawning separate timer tasks.

Awaiting a Timer (simple delay): The straightforward use is `Timer::after(duration).await;` inside your actor's loop or handler. This simply sleeps the task for that duration. However, this will pause the entire actor, meaning it won't process new messages until the timer completes. In some cases (like a periodic heartbeat actor), that's fine. But often, you want the actor to be able to **respond to either a message or a timeout**, whichever comes first.

Using `select` (Race) between Timer and Message: To wait for either an incoming message or a timeout, use the `embassy_futures::select` combinator (similar to `Future::select` or `futures::join` in other libraries). For example, suppose an actor expects a response within 1 second or else it will retry:

```
use embassy_futures::select::{select, Either};

let timeout_fut = embassy_time::Timer::after(Duration::from_secs(1));
let msg_fut = inbox.next(); // assume `inbox.next()` awaits the next message
(like receiver.receive())

match select(timeout_fut, msg_fut).await {
    Either::First((_timeout_elapsed, pending_msg_fut)) => {
        // Timer completed first, and pending_msg_fut is the message future
(still pending).
        self.on_timeout().await;
        // We might decide to loop and continue waiting for the message.
    },
    Either::Second((msg, _pending_timer_fut)) => {
        // A message arrived before timeout.
        if let Some(m) = msg {
            self.handle_message(m).await;
        }
    },
}
```

In this pattern, whichever future becomes ready first causes the select to return [47]. The other future (the one still pending) is dropped, which **cancels the timer** if it hadn't fired yet. This addresses "cancellability" – dropping a `Timer::after` future stops the waiting (Embassy's timer queue will remove it). Thus, using `select` (or Embassy's `yield_now` and friends for more complex patterns) is the idiomatic way to race a timer against other events. Embassy does not have a separate `cancel_timer(timer)` API; you cancel by dropping the future.

Composable Futures: You can compose timers with other async operations using `select`, as above, or by chaining futures. Another pattern is *timeout on a wait*: for instance, implementing a request-response actor that gives up if no response in X time. You could write a small wrapper future that awaits either a channel receive or a Timer. Alternatively, use `embassy_futures::select::select3` if you have more than two events to wait on (Embassy provides `select3` and so on for multiple-way races [48]). For clean code, you might wrap this logic in a function like `async fn wait_for_message_or_timeout(rx: &Receiver<T>, dur: Duration) -> Option<T>` which internally does the select.

Cancellation via Task Structure: Another approach is **spawning a dedicated timer task** that sends a message when time elapses. This is useful if you want the actor's main loop to remain focused on message handling and not include lots of timing logic. For example, you could spawn a one-shot task:

```
if let Some(addr) = self.address() {
    // spawn a task that waits and then notifies this actor
    let delay = Duration::from_secs(5);
    unwrap!(spawner.spawn(async move {
```

```
        Timer::after(delay).await;
        addr.notify(TimeoutEvent).await;
    }));
}
```

In this snippet, when the actor needs a timeout, it spawns a helper that waits 5 seconds then sends a `TimeoutEvent` message to the actor's inbox. If the actor receives some other input that obviates the timeout, it could ignore the `TimeoutEvent` when it arrives (or you could design it so the helper task is canceled via a flag – though often just letting it send and be ignored is fine if spurious). This technique is a form of **fire-and-forget timer**. It costs an extra task slot, but tasks in Embassy are lightweight. Just remember that any task you spawn must also be `'static` (so the address `addr` used above must be `'static`, which in an actor system it usually is).

Real-world embedded code often uses the select approach. For instance, in a multi-event scenario, you might see something like:

```
match select(connection.poll_event(), receiver.receive()).await {
    Either::First(event) => info!("Got connection event: {:?}", event),
    Either::Second(msg) => info!("Got new message: {:?}", msg),
}
```

(as shown in an example) [47]. Replacing `connection.poll_event()` with `Timer::after(...)` is conceptually the same – it's an event that might occur.

Periodic Timers: If an actor needs to do something periodically (say, send a heartbeat every 1 second), a simple pattern is:

```
loop {
    Timer::after(interval).await;
    self.do_heartbeat().await;
}
```

This works, but again it will not respond to other messages while sleeping. If that's a concern (maybe you want to handle commands and also issue heartbeats), you can restructure as two tasks or use an internal scheduling mechanism. One approach: run a dedicated "ticker" task that sends a `Tick` message to the actor every interval (similar to the one-shot timer approach, but recurring). The actor just processes Tick messages like any other. This way, the actor's main loop doesn't block on the Timer at all – timing is handled by another task. The trade-off is complexity (one more task and message).

However, many embedded devs choose the simpler route: combine message handling and timing in one loop with select. You can re-use the `select` pattern in a loop to handle periodic work:

```
loop {
    let tick = Timer::after(interval);
    match select(tick, inbox.next()).await {
        Either::First((_elapsed, inbox_future)) => {
            self.do_heartbeat().await;
            // continue loop, inbox_future is the same still-pending receive
        }
        Either::Second((maybe_msg, _tick_future)) => {
            if let Some(msg) = maybe_msg {
                self.handle(msg).await;
                // then loop back, effectively resetting the timer since we
dropped it
            } else {
                // channel closed? handle termination
            }
        }
    }
}
```

This way, the heartbeat is skipped or delayed if a message comes in (because we process the message immediately and restart the timer). Or if no message comes, the heartbeat fires at `interval` and then the loop continues. This is **composable behavior** – the actor is responsive to both inputs and time.

Cancellation Considerations: As noted, dropping a Timer future cancels it. In the select patterns above, that happens naturally. If you ever store a `Timer::after` future somewhere (not immediately await or select on it), dropping it later will cancel. Just ensure you don't await a timer longer than needed. For instance, if your actor needs to abort a wait early, you might structure your code to break out of the `.await` (which you can't easily do without select, hence the need for select/race patterns).

Another scenario is if your actor is shutting down (maybe the device is going to low-power). If the actor task is about to end (drop its inbox and finish), any pending Timer in it will be dropped naturally as the task ends. But if you spawned a separate timer task, you might need a way to cancel that. Typically, that separate task would check some shared flag or you design it such that the actor's termination is tied to a reset anyway. Full cancellation of spawned tasks can be complex (Embassy doesn't yet have a built-in cancel mechanism beyond designing tasks to monitor flags or channel closures).

Embassy Timer specifics: Embassy's timer operates on a **tickless timer wheel** integrated with the executor. It's very efficient. The time driver will sleep the MCU (if no tasks are ready) until the next timer is due [49] [50]. This means using timers liberally is fine – they do not burn CPU while waiting. When a Timer future is dropped, Embassy removes its entry from the timer queue so it won't wake up unnecessarily. All of this is internal, but the takeaway is that these patterns (racing timers, dropping them) are well-supported and won't leak or glitch.

Summary: *Use* `embassy_time::Timer::after()` *in combination with* `embassy_futures::select` *to implement timeouts and cancellable waits in actor tasks.* This allows your actor to handle either an incoming message or a timeout event in a unified way [47]. For periodic actions, either incorporate timers in your loop

with select (to remain responsive) or offload periodic ticks to a separate task that sends messages. Both approaches are used in practice, depending on complexity. The key point is that **dropping a timer future cancels it**, so whichever path in your `select` is not taken, the corresponding timer is automatically canceled (no memory or CPU cost remains) when that future is dropped.

Finally, if you need a timer that you can manually cancel outside of `select` (for instance, cancel an ongoing timeout from another context), you could use an `embassy_sync::signal::Signal` in combination with a timer task: e.g., spawn a task that waits for either a `Signal` or a Timer, whichever first, similar to select. That pattern is essentially reimplementing select with two tasks and a synchronization primitive – usually not needed when you can do the same in one task with `select`, but it's another tool (the forum discussion about implementing a button press with both GPIOTE interrupt and a hardware timer used `Signal` to coordinate which event happened first [51] [52]).

In conclusion, Embassy provides the building blocks to integrate timers cleanly. By racing timers with message reception, your actors can remain responsive and implement behaviors like *"if no message in X time, do Y"* or *"do Z every T seconds unless interrupted"* in a straightforward manner.

Error Handling and Resource Cleanup in Embassy Tasks

Designing for robustness means considering what happens when tasks end unexpectedly or when resources (like channels) are closed. In embedded systems, often tasks run forever (`-> !`) and the concept of "shutting down" is limited (usually a reset or power-off is the real end). That said, let's cover patterns for graceful error handling and cleanup:

- **Task Panics:** If an Embassy task panics (for example, due to an unwrap on an error), the default behavior (with `panic_probe` or similar) is to halt or reset the system. Unlike a desktop environment, you typically do not continue executing other tasks after a panic, because there's no OS to isolate the failure – it's all one program. Therefore, **the best practice is to avoid panics in tasks and handle errors internally**. Use `Result` returns from driver APIs and decide on a strategy: maybe log the error (using defmt or rtt logging) and attempt to recover or restart the peripheral. If a truly unrecoverable situation occurs, an embedded system might reset the microcontroller (which is effectively a cleanup). Some developers configure `defmt::panic` or `panic_halt` to simply halt so they can inspect state via debugger. But in production, you might want a watchdog to reset on panic.

There is no built-in Embassy mechanism to restart a panicked task alone. Once a task panics, it will unwind/abort according to the panic handler and likely bring down the whole executor (since we're typically in thread-mode executor, a panic will unwind into `main` unless caught, and there's no catch in no_std by default). Therefore, **ensure each actor task carefully handles errors** instead of panicking. For example, if an I^2C read fails, you might send an error message to another task or set an error state, rather than panicking.

- **Graceful Task Completion:** If an actor's job is done (say it was a one-shot task, or you have a condition to break out of its loop), the task can simply return. Embassy will mark the task storage as free (if you used `spawn()` via TaskStorage or pool) so it could be reused or just remain idle. However, since most actors are likely infinite loops, this is rare. Should a task finish, any resources it was holding (e.g., open channels, device handles) will be dropped as in normal Rust. If those

resources implement `Drop`, their cleanup will run. But note: many embedded handle types (like embassy driver instances) might not implement Drop beyond maybe disabling an interrupt; often they rely on RAII for critical sections but hardware stays initialized. There isn't typically a need to "free" hardware – at most you might put it in a low-power state.

- **Dropped Channels (Senders/Receivers):** Embassy channels do not have an explicit "close" method, but they detect when all senders or all receivers are dropped:

- If a receiver (`Channel::receiver()`) is dropped, any future `.send().await` on the corresponding sender will return an error (likely `SendError` or similar). In practice, since our actor holds the receiver and the actor runs indefinitely, this situation arises only if the actor task ends. The senders (held by other tasks) should handle this. The `send().await` future will complete with an error indicating the message wasn't delivered. The recommended pattern is to check the `Result` of `sender.send().await`. For example:

```
if sender.send(msg).await.is_err() {
    // The receiver is gone – perhaps the actor task terminated.
    // Decide on fallback: maybe log or ignore if shutting down.
}
```

  In many embedded cases, you might simply ignore it or trigger a system reset if a critical actor is gone (since that's unexpected).

- If all senders are dropped, the receiver's `receive().await` will return `None` or an error (depending on API). In Embassy's channel, the `Receiver::receive()` returns a `Future<Output=T>` (not an Option) – but internally, if no senders exist and the buffer empties, it cannot ever receive a new message, so it might wait forever. There may be an internal check for disconnection. In any case, if your design is such that an actor can stop and drop its receiver, it's wise for the actor to signal others that it is stopping (perhaps via another channel or a global flag). **Ector's approach**: it doesn't really provide a built-in actor termination; actors are expected to run forever. If an actor does need to stop, one could send it a special "Terminate" message and have it break the loop. After that, dropping the receiver is fine – any tasks still trying to send will get an error.

- **Cleaning up Resources on Drop:** If you do implement `Drop` for some of your actor's resources (say an actor that manages a sensor might implement Drop to turn off the sensor), that code will run when the actor task ends and the actor struct is dropped. With static allocation, dropping a `'static mut` that lived in a StaticCell is a bit tricky – if the task ends, technically the actor is still in static memory, but you can consider it inactive. You might not actually call `StaticCell::drop`, because StaticCell doesn't provide a way to free the memory (it's static). Instead, you might manually call some cleanup method on the actor before ending the task. For example:

```
// inside actor loop
if terminate_condition {
    self.cleanup();
```

```
    break;
}
```

where `cleanup()` does any necessary shutdown of hardware. This is one way to ensure graceful release of resources.

Another scenario: cleaning up on channel closure. If your actor is waiting on `receiver.receive()` and that unblocks with an error (meaning senders are gone), that could be your cue to break the loop and cleanup. Since Embassy channels don't return a distinct error easily, you might use an out-of-band signal (like a separate `Signal`) to indicate "stop now."

- **Using `Signal` or `Notifier` for Shutdown:** The `embassy_sync::signal::Signal` is a primitive that one task can wait on and others can signal. You could embed a `Signal` in an actor that, when triggered, causes it to exit. For instance, a supervisor task could call `shutdown_signal.signal(())` to tell an actor to stop. The actor would check `shutdown_signal.wait()` either periodically or in select with its main loop. This is a pattern for cooperative cancellation. It's simpler to implement than canceling tasks externally (since no task cancel API in Embassy yet). The downside is every actor needs to periodically check the signal.

- **Memory Leaks:** Because we use static memory, there is no traditional memory leak at runtime (no heap allocations that need freeing). The concept of a leak would be if a task stops using something but it stays allocated forever (which is just how static works). In critical systems, you usually don't worry about freeing memory – you worry more about *stale state*. For example, if an actor stopped but some global state thinks it's still active, that could be an issue. It's more a logic concern than a memory one here.

Example – Gracefully Handling a Dropped Channel: Imagine an actor that reads from a sensor and sends data to a processing task via a Channel. If the processing task goes away (drops its receiver), the sensor task's sends will start failing. We can handle this:

```
if let Err(e) = data_sender.send(reading).await {
    log::warn!("Processing task not available, dropping sensor reading");
    // Optionally break out if this means the system is shutting down:
    // break;
}
```

If such a condition is not recoverable, you might decide to reset the system or enter a safe state. In embedded, often the loss of a task is considered fatal (because tasks are usually meant to run indefinitely). So it's reasonable to simply reboot or signal a fault if an essential channel closes.

For **non-essential channels**, handling the `Err` by dropping the message (as above) is fine. The error type from `Channel::send` when the receiver is gone is typically an indicator like "Disconnected". In Embassy

0.6, you'd check if `Err == Channel::Disconnected` (if that exists) – the exact API might differ, but conceptually it's similar to how a `tokio::mpsc` returns `Err(SendError)` if the receiver is gone.

- **Panic Handling Hooks:** On some systems, you can set a custom panic handler (via `cortex-m` crate or similar) that for example toggles an LED or does a system reset after logging. This can serve as a "last resort cleanup," ensuring the system doesn't stay in a wedged state. For instance, a panic in one task might trigger a watchdog that reinitializes everything.

Resource Cleanup on Normal Stop: If you deliberately stop an actor (not via panic), ensure you turn off any hardware it was controlling. E.g., if an actor controlled a motor, send a stop command to the motor before exiting the loop. Since the actor code is under your control, just structure it such that the exit path performs these actions.

Embassy Executor Behavior: The executor will not automatically respawn tasks or do anything on a task panic – it relies on you. The Embassy executor is mostly concerned with polling tasks; if a task future ends, it marks it done. If it panicked (and you use `panic_halt` or abort), the whole execution stops. If you use a unwinding panic (very uncommon in no_std, often panic = abort is used), theoretically one task unwinding wouldn't affect others unless it propagates out of `Executor::run`. But since `Executor::run()` doesn't catch, it would propagate out and likely crash anyway. So robust design = no panics.

Summary: *In Embassy-based systems, you typically don't have dynamic resource deallocation – you design for tasks to run indefinitely and handle errors internally.* Graceful handling means checking `Result`s and reacting instead of panicking, and possibly using signals or messages to coordinate shutdown if needed. If a channel or actor is dropped unexpectedly, other components should detect it (via send errors or lack of response) and take appropriate action (log, reset, safe-state, etc.). This is very application-specific. Many embedded apps simply assume tasks don't terminate and will reset the whole system if something goes wrong (which can be a valid strategy given hardware watchdogs).

For a higher-level actor framework integration, you might implement a **supervisor** that monitors critical actors. For example, have a periodic "I'm alive" ping from each actor to a supervisor. If one stops responding, the supervisor could restart the MCU or try to re-init that actor's functionality. Without an OS, that kind of self-healing is limited, but with static actors you could conceivably call their init and spawn again (since the StaticCell can be reused once the task truly ended – `TaskStorage` can spawn again once a task is finished [53]). This is advanced and not commonly done yet, but you could experiment with it.

In summary, design your Embassy actor system under the assumption that tasks run forever and don't leak resources. Handle expected errors (like I/O errors, full queues) gracefully within the task. Use channel closure detection or signals if you need to coordinate task shutdown. And test those paths – e.g., deliberately drop a sender and ensure the receiver task behaves as you expect. Embassy gives you the tools (channel `Result`, signals, etc.), but it's up to you to implement the policy for what to do on those events.

Performance: Embassy Channels vs. Tokio Channels

It's useful to compare the characteristics of Embassy's synchronization primitives (like Channel) with their Tokio counterparts, especially since you want to match Tokio's behavior in your actor system.

Memory Footprint: Embassy's channels are **statically allocated with a fixed capacity**, whereas Tokio's channels (e.g. `tokio::sync::mpsc`) allocate on the heap (dynamic memory). For an embedded target with no global allocator, Embassy's approach is essential – you pay upfront memory cost but no ongoing allocations. Tokio's channel on a desktop or OS allocates heap memory for the queue (bounded channels allocate a buffer once, unbounded channels allocate on demand). In terms of usage in an actor system: - **Embassy Channel:** You decide on a capacity N for each mailbox. If N is too low and messages arrive faster than processed, `.send().await` will simply wait (no overflow unless a `.try_send` variant is used). If N is higher than ever needed, you just waste a bit of RAM. This trade-off is well worth the determinism in embedded. You can confidently say "this mailbox uses X bytes" – for instance, a channel of 32 `Increment` messages (from the earlier example) might use only a few dozen bytes. - **Tokio Channel:** On the PC, memory is more plentiful, and Tokio can dynamically grow an unbounded channel. But that comes with the risk of unbounded memory usage if messages flood. In embedded, we avoid that by design. So your actor system on Embassy will naturally enforce backpressure due to bounded channels.

Throughput and Latency: Tokio's channels are highly optimized for high throughput on multi-core systems. Embassy's channels are optimized for simplicity and low overhead on microcontrollers. In practice: - For small message rates (e.g., sensor readings at 100 Hz, or user commands at a few per second), both Embassy and Tokio channels will handle these easily with negligible CPU usage. - If you theoretically push many thousands of messages per second, a PC with Tokio might achieve higher absolute throughput (owing to more CPU power and possibly lock-free algorithms). But an embedded device running at, say, 120 MHz likely cannot handle extremely high message rates across many tasks due to CPU limits anyway. Embassy's channel uses a critical-section or atomic operation per send/receive, which on a microcontroller is maybe a couple of microseconds overhead. That's usually fine; if an actor can process, say, 50k messages/sec, the limiting factor might be the actor's work, not the channel mechanics. - **Benchmark Data:** There isn't a public, apples-to-apples benchmark of Embassy vs Tokio MPSC channels in 2025 that we can cite. However, qualitatively, Embassy's developers note the executor and sync primitives are designed for efficiency in their domain. The Embassy book points out that compared to an RTOS, the async executor can be more efficient in scheduling [49]. For channels specifically, anecdotal evidence from users suggests that the overhead is low. Eloy Coto's blog mentions "the computational overhead isn't that high" for Embassy channels [23] – implying they are plenty fast for typical usage.

Tokio's channel, on the other hand, might use a more complex mechanism (internally using linked lists or atomic ring buffers). The overhead per message might be on the order of tens of nanoseconds on a PC (very small). On an embedded, Embassy's might be a few hundred nanoseconds to a few microseconds depending on clock speed. These differences are insignificant unless you are truly micro-optimizing.

Scheduling Differences: One key difference: Tokio channels can wake up a receiving task potentially on a different CPU thread. Embassy's executor is usually single-thread (unless you set up multiple executors for different priorities or cores). This means in Embassy, when a message is sent, the receiving task will be woken on the same event loop and will run when it's its turn (after current task yields). Tokio, running on OS threads, could schedule receiver on another core in parallel if available. So **Tokio can achieve parallelism**, whereas Embassy (on a single-core MCU) obviously can't – everything is cooperative. In practice, for an embedded actor system, this just means you don't get preemptive parallel processing of messages. Each actor processes sequentially and relies on awaits to yield. This is by design for embedded determinism.

Power and Performance: A noteworthy point for embedded is that Embassy's approach can be more power-efficient than a constantly running loop or RTOS tasks polling. When channels are empty and no

timers pending, the executor will idle the CPU (`WFI` instruction or similar) [49] [50]. Tokio on a PC will also block threads when idle, but power isn't usually the first concern there. The bottom line is both systems only use CPU when there's work (messages or timers), which is good.

Comparative Summary:

- *Memory:* Embassy – fixed, static, zero heap. Tokio – dynamic, requires heap (which your embedded target likely doesn't even have). **Embassy wins for embedded memory control.**
- *Throughput:* Both are fast. Tokio might scale better on multi-core big iron; Embassy is plenty fast for microcontroller-scale workloads. You won't bottleneck on the channel mechanism before you hit other limits.
- *Latency:* Embassy channel latency is on the order of task switching latency (a few microseconds). Tokio channel latency can be similarly low, but on OS it might also depend on thread scheduling (tens of microseconds to wake a thread perhaps). On a microcontroller, since everything is in one loop, the latency from send to receive is typically just one executor tick – often sub-millisecond. Hard real-time latency is more predictable in Embassy because there's no OS jitter, but you must ensure no task monopolizes the executor.
- *API Surface:* Your goal is matching API. Fortunately, the usage of your actor system will probably be similar on both: e.g., an `Address<Msg>` that internally uses either a tokio::mpsc or an Embassy Channel. The semantics differ slightly (Tokio's send returns a Result indicating if the receiver is closed; Embassy's send returns `()` on success and likely would return an error if closed). You can abstract that so the actor code doesn't care.

One more note: **Embassy's** `PriorityChannel` (if you ever consider it) can provide prioritized message queues [54], which Tokio doesn't have out-of-the-box. It's probably not needed unless you have critical vs non-critical messages in one actor.

In practice, no performance complaints have surfaced in the community about Embassy channels for actor-like use. For example, the ector framework successfully uses Embassy channels to deliver potentially frequent messages (like rapid sensor interrupts) to actors and is entirely no-alloc and efficient.

If you are curious about raw throughput: A microcontroller at 100 MHz could likely send/receive on a Channel well over 100k times per second (just an estimate, considering a context switch might be ~1-2 µs). Tokio on a modern PC can do millions per second. But your embedded app probably doesn't need anywhere near that message rate.

Conclusion: *Embassy's channels are highly suitable for embedded actor messaging, with deterministic memory usage and sufficient performance. Tokio's channels excel in a different environment with dynamic memory and potential multi-threading.* Your actor system can present the same high-level API (e.g., an `ActorContext.spawn()` or `Addr.notify()`) and internally use whichever channel type is appropriate. Just document that on embedded it's fixed-capacity (and maybe provide the capacity as a const generic or associated const in your actor definition so it's clear).

By following Embassy's patterns, you'll get an actor system that is close in behavior to the Tokio version, with the main differences being in resource limits rather than functionality. And as a plus, you won't be using any heap, making the system more predictable and RT-friendly [23].

Sources:

1. Embassy executor and StaticCell usage (Embassy 0.6 docs and issues) – e.g., using `StaticCell` for executor and tasks [28] [6].
2. Embassy channel vs pipe and examples – Embassy documentation and community examples [14] [23].
3. Embedded Rust actor frameworks – Ector crate documentation and macros [9] [10].
4. Embassy futures and select usage – Embedded Rust blogs and Embassy book [47] [27].
5. Real-world code from ESP32/STM32 projects using Embassy (for static init patterns) [32] [7].

---

[1] [4] [13] [42] [43] GitHub - embassy-rs/static-cell: Statically allocated, runtime initialized cell.
https://github.com/embassy-rs/static-cell

[2] [5] [6] [12] [31] [32] [33] [34] Assistance with LifeTimes `embassy` + `ledc` example · Issue #870 · esp-rs/esp-hal · GitHub
https://github.com/esp-rs/esp-hal/issues/870

[3] Embedded rust - comparing RTIC and embassy | willhart.io
https://willhart.io/post/embedded-rust-options/

[7] [8] [Help] SPI with DMA transfert with embassy. · Issue #171 · almindor/mipidsi · GitHub
https://github.com/almindor/mipidsi/issues/171

[9] [10] ector-macros 0.1.0 - Docs.rs
https://docs.rs/crate/ector-macros/0.1.0/source/README.md

[11] [37] [38] [53] TaskStorage in embassy_executor::raw - Rust
https://docs.embassy.dev/embassy-executor/git/cortex-m/raw/struct.TaskStorage.html

[14] [15] [18] [19] [20] [22] embassy_sync::channel - Rust
https://docs.embassy.dev/embassy-sync/git/default/channel/index.html

[16] [17] [25] Pipe in embassy_sync::pipe - Rust
https://docs.embassy.dev/embassy-sync/git/default/pipe/struct.Pipe.html

[21] [23] [24] [27] [47] Practical Embedded Rust Development Guide with Embassy
https://acalustra.com/embedded-rust-development-tips-with-embassy.html

[26] Sharing Data Among Tasks in Rust Embassy: Synchronization ...
https://blog.theembeddedrustacean.com/sharing-data-among-tasks-in-rust-embassy-synchronization-primitives

[28] [30] Executor in esp_hal_embassy - Rust
https://docs.espressif.com/projects/rust/esp-hal-embassy/0.7.0/esp_hal_embassy/struct.Executor.html

[29] [40] Embassy Book
https://embassy.dev/book/

[35] [36] esp-embassy-config — embedded dev in Rust // Lib.rs
https://lib.rs/crates/esp-embassy-config

[39] Rust Networking with the Raspberry Pi Pico W - Murray Todd Williams
https://murraytodd.medium.com/rust-networking-with-the-raspberry-pi-pico-w-002384a5954b

41 embassy_executor - Rust - Docs.rs
https://docs.rs/embassy-executor

44 embassy/examples/rp/src/bin/blinky_two_channels.rs at main - GitHub
https://github.com/embassy-rs/embassy/blob/main/examples/rp/src/bin/blinky_two_channels.rs

45 [PDF] Asynchronous Development
https://pmrust.pages.upb.ro/assets/files/ma-04-917196671dc8c685d5b3cc7a22327708.pdf

46 esp-embassy-wifihelper - Lib.rs
https://lib.rs/crates/esp-embassy-wifihelper

48 mqtt connect failed cause “Guru Meditation Error: Core 0 panic'ed ...
https://github.com/esp-rs/esp-idf-svc/issues/362

49 50 Rust: Embedded, Async, all the way!
https://ctron.github.io/eclipsecon-2022-rust-embedded/

51 52 Embassy: Implement Button Task - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/embassy-implement-button-task/107520

54 PriorityChannel in embassy_sync::priority_channel - Rust - esp-rs
https://docs.esp-rs.org/esp-idf-hal/embassy_sync/priority_channel/struct.PriorityChannel.html