ChatGPT

Detecting `async` Blocks and Generating Dual Sync/Async Code in a Proc Macro

Detecting Top-Level `async { ... }` Blocks with Syn

Rust's `syn` parsing library makes it straightforward to detect if a user has provided an **asynchronous block** at the top level of an expression. The AST enum `syn::Expr` has a variant `Expr::Async` specifically for an `async { ... }` block [1]. In a procedural macro, you can parse the user's code block as a `syn::Expr` and then pattern-match on that enum:

- If the parsed expression is `Expr::Async(ExprAsync { .. })`, it means the user wrote a top-level `async { ... }` block [1]. This is your signal to generate the *asynchronous* code path.
- If the expression is a normal block (`Expr::Block`) or any other expression, then no top-level async was used, and you can treat it as *synchronous* code.

Top-level vs. nested async: Focusing only on top-level `async` blocks keeps the logic simple. You do **not** need to analyze nested `.await` calls or function internals – Rust won't allow an `.await` outside of an async context anyway. By requiring the user to explicitly wrap async logic in an `async { ... }` block, the macro can reliably detect asynchronous intent without digging through every call. This approach avoids false positives/negatives and keeps compile-time analysis lightweight.

Example: If a user writes `entry: async { do_async_work().await; }` in the DSL, `syn` will produce an `Expr::Async` node. If they write `entry: { do_sync_work(); }`, you'll get an `Expr::Block`. Your macro's Parse implementation or procedural function can simply check for these variants (e.g. using `if let syn::Expr::Async(async_expr) = user_expr { ... }`). This established pattern is straightforward and **ensures you only respond to an explicit async block** (no guessing about what a function call might do internally).

Generating Separate Sync and Async Code Paths (Compile-Time)

Once you can detect an async block in the input, you can **generate entirely different code** for the sync vs async cases. The goal is to push all branching to compile-time (in the macro) rather than runtime. This may make the macro more complex, but it preserves zero runtime overhead for the sync-only case.

Crate patterns for dual implementations: One instructive example is the `maybe_async` proc-macro crate. It allows library authors to write one API that can be toggled between sync and async versions at compile time. Under the hood, `maybe_async` defines separate implementations for sync and async and uses attributes to include/exclude them. For instance, a function or impl block can be annotated, and the macro generates two versions – one with `async` and one without. In the RSpotify library, they use `#[maybe_async]` on traits and impls, plus helper attributes `#[sync_impl]` and `#[async_impl]` to provide distinct definitions for each mode [2] [3]. For example, a trait method can have an async implementation for an *async HTTP client* and a separate sync implementation for a *blocking HTTP client*:

```
#[maybe_async]
trait HttpClient {
    async fn get(&self) -> String;
}

#[sync_impl]
impl HttpClient for UreqClient {
    fn get(&self) -> String { /* ...synchronous GET... */ }
}

#[async_impl]
impl HttpClient for ReqwestClient {
    async fn get(&self) -> String { /* ...async GET... */ }
}
```

[3] [4]

Here the macro emits two versions of the `get` method depending on a feature flag (one awaits inside, one doesn't). The key idea is **the macro expands to completely separate blocks of code** for each mode. There's no runtime `if/else` to choose sync vs async – it's decided at compile time by the macro (or Cargo feature in this case). This approach yields *zero cost abstractions*: when compiled in sync mode, none of the async machinery is present at all [5].

In your use-case (a statechart DSL), you can apply a similar strategy per user invocation instead of per feature. If the parsed statechart contains no `Expr::Async`, generate the "pure sync" implementation (with synchronous trait impls and function bodies). If an async block is detected, generate an alternate implementation that uses async futures and `.await` as needed. The user doesn't toggle a feature – the macro itself decides based on their input.

Pattern in practice: Many libraries choose compile-time branching like this to avoid overhead. The popular `async-trait` macro, for example, transforms async functions in traits into a boxed future return type to make them dyn-safe. However, that convenience comes at the cost of heap allocation (boxing) and a vtable call on every call [6] [7]. In a no-std environment, and for maximal performance, you want to avoid that. So instead of boxing, prefer **static dispatch**: generate concrete types for futures and use generics or associated types to name them, so the compiler knows the exact return type for each async handler at compile time. This leads into how to model the handlers.

Modeling Sync vs. Async Handlers with Zero Overhead

To keep runtime overhead zero, the generated code for handlers should avoid any kind of dynamic dispatch or runtime branching. Each handler should either be a normal function (sync case) or an `async fn`/future (async case), with the decision made at compile-time. Some **best practices** and patterns to achieve this:

- **Use associated types or generics for futures:** A proven pattern (now possible on stable Rust) is to use **Generic Associated Types (GATs)** to allow an async handler without boxing. In your actor/statechart context, define a trait with an associated future type that can borrow the state machine. The `lit-bit` library's actor trait is a great example: it uses a GAT `type Future<'a>` associated with the trait, and a `handle(&'a mut self, msg: Event) -> Self::Future<'a>` method [8]. The trait is implemented differently for sync vs async cases, but always with static dispatch. For a **sync handler**, you can set `type Future<'a> = core::future::Ready<()>` (a future that is immediately ready) and simply return `core::future::ready(())` in the handler [9]. This introduces no heap allocation or scheduling – it's literally just returning a `Ready(())` future, which is optimized to a no-op. For an **async handler**, you can leverage Rust's ability to infer an opaque `impl Future` for the returned associated type. In the trait impl, set `type Future<'a> = impl core::future::Future<Output=()> + 'a` and implement `handle()` by writing an `async move { ... }` block (or calling an async helper function) [10]. The compiler will generate a concrete anonymous future type for that async block, and that type satisfies the `Future` trait. Again, no boxing is needed – the future is a static type.

- **Separate sync/async trait implementations:** If it's too complex to use a single trait for both (especially in a no_std context), another approach is to define **two traits or two versions of the API** – one for sync, one for async – and implement one or the other (or both) depending on the input. The macro could emit a trait impl for a `SyncStateMachine` and/or `AsyncStateMachine` trait. However, this would complicate usage (the user would have to choose the right trait to use). The GAT approach above is nicer because it *unifies* sync and async handling in one trait interface, and the presence or absence of `async` in the user's code simply affects the associated `Future` type. The lit-bit actor system demonstrates this: it defines a single `Actor` trait that works for both sync and async handlers, using `type Future<'a>` as a hook [11] [9]. Synchronous code can be written normally (and in fact can even be written inside an `async move` block with no `.await` – the compiler will optimize it to a direct call) and remains zero-cost [12]. As soon as you introduce an `.await` (meaning you wrote an `async` block or async function), you're then returning a real `Future` that the runtime will poll [10]. But crucially, that future is a concrete, statically known type – no trait objects.

- **No runtime flags:** Avoid designs where the statechart carries a runtime flag like `is_async: bool` and checks it on each event. Instead, **bake the choice into the type system**. For example, you might generate an internal enum of futures for mixed cases, but even that enum will be a concrete type with a `Future` impl (so it's still static dispatch). If mixing sync and async in one state machine, one technique is to return an `enum HandlerFuture<'a>` that implements `Future<Output=()>`, with variants like `Immediate(())` for sync and `Pending(Fut<'a>)` for async. This introduces a small branch in the future's `poll`, but it's a fixed, compile-time branch (not a function call) and can be optimized. However, many times it's acceptable to simply make the *entire* event handler function async if any part needs to be awaited – the overhead for purely sync events in that case is that they still go through a trivial `Poll::Ready` path in the future state machine, which is usually negligible. The primary goal is to **avoid heap allocation or trait-object indirection**, which these patterns achieve.

Examples of Crates and Frameworks Supporting Both Sync & Async

Several Rust libraries have confronted the sync/async duality and can serve as inspiration:

- **State machine / actor frameworks:** Your own project *lit-bit* is implementing this dual sync/async approach at the macro level in a statechart DSL. It builds on the Actor trait pattern described above. Other actor frameworks like Embassy's `ector` or RTIC also support no_std async by avoiding allocations – typically by requiring futures to be declared with static lifetimes or using cooperative scheduling. While not proc-macro based, they show it's feasible to mix sync and async tasks in embedded. **Async state machine generators** (e.g. the older `state_machine_future` crate) turned state machines into futures, but in modern Rust you can often just use async/await directly as the "state machine". The `async-hsm` crate is another example that models state machines as async functions, though it's more of a manual pattern than a code-generating macro.

- `async-trait` **crate:** This popular attribute macro lets you write `async fn` in traits for ergonomic code, at the cost of a runtime allocation. It demonstrates generating different code under the hood: an `async fn` in a trait is transformed into a regular sync fn that returns `Pin<Box<dyn Future>>` (plus some glue to pin and poll it) [7]. While not zero-cost, it's a clear example of a proc-macro altering trait bounds and method signatures based on async usage. Your macro should improve on this by generating **separate, allocation-free code paths** for async vs sync. In essence, you're doing a specialized form of what `async-trait` does, tailored to your DSL.

- `maybe_async` **crate:** As discussed, it uses a procedural macro to conditionally remove `async`/`await` and even duplicate items with different suffixes for each mode [13] [14]. This is a more extreme solution, mainly to support building one library with both sync and async flavors simultaneously (which normal feature flags struggle with). The design shows the lengths one can go to avoid runtime costs: they literally generate two structs, two trait impls, etc., one sync and one async, so that each is as efficient as possible in its domain [15] [14]. In your case, you likely don't need to duplicate the entire state machine type – you can integrate the duality internally – but it's useful to know this pattern exists for reference. The **lesson** is that it's better to increase macro complexity (even generating duplicate code or multiple impl blocks) than to introduce branches in the finished binary that select between sync/async at runtime.

- **Async DSLs in frameworks:** Some higher-level frameworks allow both sync and async handler functions. For example, web frameworks (Rocket, Actix, etc.) let you mix sync and async request handlers, but they typically do so by **overloading** or using separate macros/attributes (e.g. `#[get]` vs `#[get] #[tokio::main]` combinations) rather than automatically detecting code content. Since you have full control inside the `statechart!` macro, automatic detection with `syn::Expr::Async` is more direct in your scenario.

Avoiding Pitfalls and Runtime Surprises When Mixing Sync & Async

When designing the macro, a few safeguards and patterns will help ensure a smooth API without surprising behavior:

- **Document the execution model clearly:** If a user mixes sync and async handlers in one state machine, they need to understand that the statechart as a whole will execute events asynchronously (one at a time, awaiting completion of each). There's no *true* parallelism introduced – your actor model still processes one event at a time – but an event that hits an `async {}` handler will be processed cooperatively (yielding to the executor) rather than blocking. This should be made clear to avoid assumptions that an event handled by sync code will always complete immediately. In practice, the difference is usually minimal (the next event still won't start until the current one's future is done, per your actor's guarantees [16]), but it matters for understanding performance on embedded: an async handler might wait on, say, I/O or a timer, during which the thread/executor can do other things. As long as only one event per actor is active, *determinism* is maintained [17].

- **Require explicit `async` for async behavior:** Your approach already does this by looking for an `async { ... }` block. This is a good design – it means the user *opts in* to async semantics explicitly at the DSL level. There's little chance of accidental async usage without noticing (since they'd have to write the `async` keyword). This avoids subtle bugs like accidentally making a handler async when it wasn't intended. It also prevents the user from trying to call `.await` on something without the proper context – if they do, the Rust compiler will error out unless they wrap it in `async {}`. So your macro doesn't have to catch that; the language ensures that async operations are only used in an async context.

- **Compile-time feature gating for no-std:** Since you target `#![no_std]` environments, you might conditionally compile or emit different code depending on whether an allocator is available. For example, if no `alloc` and the user writes an async block, you'll generate code using only `core::future::Future` and require an executor like Embassy (which uses no heap by default). If `std` is available, you might integrate with Tokio or allow the more ergonomic but heap-using `AsyncActor` trait (like a boxed future convenience) [18]. Ensure that using an async handler without an allocator remains possible – which it will if you stick to core futures and pinning on the stack (GATs help here). It sounds like you already have conditional mailbox implementations and spawn functions for Embassy vs Tokio [19] [20]. The macro can piggy-back on those: e.g., it could emit an error or warning if an async handler is used without enabling the feature that provides an async runtime, or automatically toggle the appropriate feature.

- **No hidden blocking:** One "runtime surprise" to avoid is blocking inside an async context. Encourage or enforce non-blocking design for async handlers (e.g., don't call a blocking function like a busy delay inside an `async` handler without at least marking it). In embedded, blocking in async can starve other tasks. While your macro can't easily detect "this call is blocking", you can provide guidance in documentation: if the DSL allows calling arbitrary functions in handlers, advise users to keep sync handlers CPU-bound but quick, and use async handlers for anything that might wait (delays, I/O) so it yields to the executor. This will naturally happen if they use async APIs for those operations.

- **Testing sync vs async outputs:** It's a good idea to test that when no async blocks are present, the expanded code is literally the same as a purely synchronous statechart (no extra future glue). Conversely, test that an async handler yields code that compiles and runs under an executor. Having these as separate tests (maybe behind features like `test-sync` and `test-async`) can ensure your macro generation stays on track. This way you won't unknowingly introduce overhead into the sync case while adding async support.

- **Mixing within one statechart:** If you allow some states or transitions to use async and others to remain sync, make sure your generated code handles each correctly. Likely, as discussed, you'll make the overall event processing function `async` if any part is async (because you can't `.await` otherwise). Within that async event handler, purely sync actions are just normal code (they execute immediately). This is fine – they'll just execute and not suspend. One thing to double-check is that entry/exit/transition actions run to completion in the correct order even if some are async. For example, if an exit action is sync and the subsequent entry action is async, you'll want to ensure the exit action is performed fully before awaiting the entry (which it will if you write it in sequence in the async block). Essentially, the macro should sequence actions in a logical order with `await` at points where async boundaries exist. By keeping those semantics straightforward (and mirroring what a purely sync statechart would do), you avoid surprising the user.

In summary, **macro-driven detection and code generation** is the idiomatic way to support a dual sync/async API in Rust without compromising on performance. You choose complexity in the proc-macro over complexity in the runtime. Use `syn` to identify `async` blocks in the DSL input [1], then emit code that either implements all handlers as normal functions (for sync) or as `async fn`/futures (for async). Leverage patterns from the ecosystem like GAT-based async traits for no_std (see the `Actor` trait example in lit-bit, which uses stack-pinned futures with no allocation [11] [9]). Draw inspiration from crates like `maybe_async` (conditional compile-time dual implementations) and `async-trait` (proc-macro transforming APIs based on async usage) – but improve on them by ensuring *no* runtime dispatch and no feature toggles needed per state machine. By carefully designing the macro output, you'll maintain **identical performance and binary code** for sync-only statecharts and introduce async futures only when the user explicitly requests them, which aligns perfectly with zero-cost abstractions and embedded/no-std requirements.

---

[1] Expr in syn - Rust
https://docs.rs/syn/latest/syn/enum.Expr.html

[2] [3] [4] [5] The bane of my existence: Supporting both async and sync code in Rust | NullDeref
https://nullderef.com/blog/rust-async-sync/

[6] [7] async_trait - Rust
https://docs.rs/async-trait/latest/async_trait/

[8] [9] [10] [11] [12] [16] [17] [18] mod.rs
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/lit-bit-core/src/actor/mod.rs

[13] [14] [15] async and sync in the same program · Issue #6 · fMeow/maybe-async-rs · GitHub
https://github.com/fMeow/maybe-async-rs/issues/6

[19] [20] 2025-05-25.md
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/docs/PROGRESS/2025-05-25.md