ChatGPT

Structuring a Multi-Target `#![no_std]` Rust Workspace

Designing a Rust workspace that targets both embedded (`no_std`) and hosted (`std`) environments requires careful isolation of development-only dependencies. The goal is to ensure that heavy dev tools (e.g. `criterion`, `proptest`, `tokio`) **never get pulled in when building for embedded targets**, while still allowing their use for testing, simulation, and benchmarking on the host. Below we outline best practices for workspace layout, dependency management, example organization, and build automation that have been adopted in the Rust embedded ecosystem.

Isolating Dev Dependencies from Embedded Builds

By default, **Cargo will compile dev-dependencies when building tests, examples, or benches**, even for a `#![no_std]` crate [1]. This means if your library crate has a dev-dependency on, say, `criterion` or `tokio`, then running `cargo build --examples` (or any command that builds dev targets) for an embedded target will try to compile those crates – which will likely **fail** or bloat the build. In fact, Cargo's current behavior is that *"dev-dependencies are NOT used when building the lib, but they are enabled when building examples... when you build any example, all the dev-dependencies are enabled."* [2]. This can accidentally introduce a dependency on `std` or other unwanted features in an embedded build.

To prevent this contamination, the core strategy is to keep dev-only dependencies out of the embedded crate. Effective approaches include:

- **Use a dedicated workspace crate for tests/benches**: Instead of listing heavy dev crates under `[dev-dependencies]` of your `no_std` library, create separate crate(s) in the workspace for them. For example, you might have `mylib-core` (the `#![no_std]` library) and then `mylib-tests` or `mylib-benches` crates that depend on `mylib-core` (with `std` enabled) and include things like `proptest`, `tokio`, or `criterion` in their normal `[dependencies]`. This way, building `mylib-core` for an embedded target never even sees the dev stuff – those are only pulled in when you explicitly build the test or bench crate on a host. Many projects use this pattern, e.g. naming crates like `myproject-tests` or `myproject-benches` and marking them `publish = false` in Cargo.toml so they are not published.

- **Move examples that require `std` into separate crates or binaries**: If your library needs example programs for both embedded and host, keep them apart. You can put **device-targeted examples** (which use `no_std` + embedded HAL) in the `examples/` of the embedded crate, but ensure they only use `no_std` dependencies. Any example that requires `std` or dev crates (for simulation or complex config) should live in a separate example crate or binary that depends on your library as a normal dependency. This recommendation is echoed by the community: *"it's better to move the examples (at least those needing non-trivial configurations) into separate packages, because of the limitation of dev-dependencies."* [3] By doing so, a `cargo build` for the embedded target won't pull in unnecessary dev deps. For instance, you might have an `examples/` directory at the workspace level with sub-crates like `blinky-std-sim/` (a PC simulator using `std`) and `blinky-embedded/` (an embedded `no_std` binary), rather than using the library's inline examples for both.

- **Leverage target-specific dev-dependency sections**: Cargo allows conditioning dev dependencies on target via the `Cargo.toml` target table. You can specify dev dependencies only for host platforms so they are ignored for bare-metal targets. For example, in your library's Cargo.toml you could do:

```
[target.'cfg(not(target_os = "none"))'.dev-dependencies]
tokio = "1"
criterion = "0.4"
```

This means for any target where `target_os != "none"` (i.e. a hosted OS like Linux, Windows, etc.), these dev deps apply, but for embedded targets (which usually have `target_os = "none"` in their triple) they will **not** be included. Cargo's documentation confirms that *"you can have target-specific development dependencies by using* `dev-dependencies` *in a target section"* [4]. This approach keeps dev crates out of the build for embedded architectures entirely. (Do note that target-specific dev-deps must include a `version` since dev deps are ignored for publishing [5].)

- **Use the 2021+ Cargo feature resolver**: Ensure your workspace uses the new feature resolver (Edition 2021 or set `resolver = "2"` in Cargo.toml). The new resolver prevents dev-dependency features from leaking into normal builds [6]. In other words, features enabled by dev crates won't activate features in your regular deps unless you are actually building tests/examples. This helps avoid the classic issue where a dev-dep accidentally enabled the `std` feature of some transitive dependency, breaking no_std builds [7] [8]. With resolver 2, a plain `cargo build` for an embedded target will truly ignore dev-dependencies and their features. (Dev deps will still be compiled if you explicitly build tests or examples, of course.)

Bottom line: Keep the embedded-facing crate as "clean" as possible. Offload anything that isn't needed on the device into other crates or conditional sections so that an embedded build sees only `no_std` compatible code.

Workspace Layout for Multi-Target Projects

Organizing your workspace into multiple crates is a maintainable way to support both embedded and host targets. A common pattern in larger embedded Rust projects (like Embassy, RTIC, etc.) is to **split hardware-specific code from portable logic**. Ferrous Systems, for example, recommends using even nested workspaces to clearly separate target-specific and host-specific components [9]. In our context (a library crate), a simpler flat workspace might suffice:

- **Core library crate (** `mylib-core` **)**: This is your primary library marked `#![no_std]`. It contains the core functionality that runs on embedded targets. This crate should have *no mandatory* `std` *dependency*. If needed, provide an **optional** feature (e.g. `"std"`) to enable std support. By default (with default features off), it remains no_std. For example, in `mylib-core/Cargo.toml`:

```toml
[package]
name = "mylib-core"
# ...

[features]
default = []            # no_std by default
std = []                # enable use of `std`
```

And in `src/lib.rs`:

```rust
#![no_std]
#[cfg(feature = "std")]
extern crate std;
```

This allows the crate to link against the standard library when `std` feature is on, but remain no_std otherwise. Use `cfg(feature = "std")` in the code for any std-specific functionality (e.g. implementations or modules that should only be available with std). This conditional compilation is the idiomatic way to have one crate serve both environments. For instance, if you have an API that uses `std::io` traits or spawns threads (Tokio), put it behind `cfg(feature = "std")` or behind a separate feature flag for that capability.

- **Testing crate (**`mylib-tests`**)**: A host-only crate that **depends on** `mylib-core` (with `features = ["std"]` to enable any std support in core). This crate can contain integration tests or use the standard test harness. You might make it a binary or library crate with tests in the `tests/` directory. Include heavy test deps here (e.g. `proptest`, `quickcheck`, etc.) as **normal** dependencies (not dev-deps), since this crate itself is never built for the target device. For example, `mylib-tests/Cargo.toml` might have:

  ```toml
  [dependencies]
  mylib-core = { path = "../mylib-core", features = ["std"] }
  proptest = "1.0"
  ```

  Then you can write extensive property-based tests, simulation tests (maybe using `tokio`), etc., all within this crate. Running `cargo test -p mylib-tests` will exercise them on the host. Meanwhile, `cargo build -p mylib-core --target thumbv7m-none-eabi` will *not* involve any of these test crates or their deps.

- **Benchmarks crate (**`mylib-bench`**)**: Similar to the tests crate, you can create a separate package for benchmarks. Criterion (and others) require `std` and bring in large deps, so isolating them is wise. `mylib-bench` would depend on `mylib-core` (with std enabled) and include `criterion` in its `[dependencies]`. You can put Criterion benchmarks in this crate's `benches/` directory or simply call Criterion from a `main.rs`. Running `cargo bench -p mylib-bench` will run the benchmarks on the host. This separation ensures that CI or users building the core library for embedded are not forced to compile Criterion (which can save a lot of time and avoid compiler errors on no_std targets). In fact, some projects exclude the benchmark crate from the default workspace build – for example, the Ruma project noted that the way to avoid compiling Criterion in CI was to move benches to a separate crate and exclude it from the workspace by default [10].

- **Examples**: As discussed, prefer to organize examples as separate small crates or binaries. In a simple case, you might have an `examples/` directory at the workspace root with subdirectories for each example. For instance, `examples/blinky_no_std/Cargo.toml` (which depends on `mylib-core` and an embedded HAL for a real board) and maybe `examples/blinky_std_sim/Cargo.toml` (which depends on `mylib-core` with std and simulates the device logic on your PC, possibly using `tokio` for timing). Each of these can be a `[package]` in the workspace. This way, you **build embedded examples with** `cargo run --release -p blinky_no_std --target thumbv7m-none-eabi`, and build or run the PC simulation with `cargo run -p blinky_std_sim` on your host. They have separate dependency sets appropriate to each environment.

- *Alternative:* If you prefer using Cargo's built-in example system (i.e. `src/bin` or `examples/` within a crate), then restrict which examples are built for which target. You can use the `required-features` field in Cargo.toml for examples to ensure they only compile when appropriate features are enabled. For instance, an example that needs `std` could be declared with `required-features = ["std"]` so it won't even attempt to build when `--no-default-features` (no std) is used. This can prevent compile errors if someone does `cargo build --examples --no-default-features --target thumbv7m-none-eabi`. Keep in mind, however, that **dev-dependencies apply globally to the crate**, not per example – so even with `required-features`, Cargo may still resolve all dev-deps if any example is built [2]. Thus, truly splitting into separate example crates (which can have their own Cargo.toml and dependencies) is the cleanest solution to avoid dev-dep leakage.

- **Workspace Cargo.toml**: Your top-level `Cargo.toml` (virtual workspace) should list all the member crates (`mylib-core`, `mylib-tests`, `mylib-bench`, `blinky_no_std`, etc.). You can also define `default-members` to control which packages build by default. For example, you might set `default-members = ["mylib-core"]` (and maybe your most important example) so that a bare `cargo build` in the workspace only builds the core library. This prevents CI or newcomers from accidentally building all tests/benches/examples unless they intend to. (They can always build them explicitly or run `cargo build --workspace` to include everything.) Using `default-members` is a way to keep CI lean – e.g., only build the library and perhaps critical examples, while benches or extensive tests are opted into separately.

In summary, a workspace approach gives you clean modular separation: **the embedded-facing crate stays slim and** `no_std`, and all development and host-specific code lives in other crates that you only invoke on demand. This structure also plays nicely with ecosystem tools – for instance, you can have independent `Cargo.toml` configurations, separate `#[cfg]` settings, and more targeted CI steps for each part.

Conditional Compilation for `std` vs `no_std`

Within your crates, use **feature flags and `cfg` attributes** to differentiate code for embedded vs. host. As noted, the typical pattern is an opt-in `"std"` feature for libraries. By default the crate is `no_std` (ensuring it can compile for targets like `thumbv7m-none-eabi`), and enabling the `std` feature unlocks additional capabilities (like implementations of traits that require `std::error::Error`, or integration with host-only libraries).

Some tips for idiomatic conditional compilation:

- **Use `#[cfg(feature = "std")]`** to gate any code that uses the standard library. For example, if you have a debug or simulation module that uses file I/O or threads, wrap it as:

```
#[cfg(feature = "std")]
pub mod pc_simulator {
    // ... code that uses std or Tokio, etc.
}
```

Consumers of your library can enable the `"std"` feature when running on a desktop, and disable it (default) for embedded. This way, `cargo build --no-default-features --target your-embedded-target` will not even compile that module. This approach extends to dependencies as well – e.g., you can make Tokio an optional dependency activated by a feature. In Cargo.toml:

```
[features]
std = ["tokio"]  # (if Tokio is only used with std)

[dependencies.tokio]
version = "1.28"
optional = true
```

Then any code using `tokio` is inside `#[cfg(feature = "std")]` or perhaps a more specific feature like `async_std` etc. This ensures no Tokio on embedded builds unless explicitly allowed.

- **Use `alloc` for heap without std**: If your no_std code needs heap allocations (e.g., for property tests or data structures), consider using the [`alloc`] crate. On stable Rust, you can do `extern crate alloc;` under a feature (often the `"alloc"` feature, or include it under `"std"` since std implies alloc). This allows using `Vec`, `String`, etc. in no_std context (provided the target has an allocator or you use a global allocator like `wee_alloc`). Some dev crates like `proptest` offer an `"alloc"` feature to run in a no_std environment with allocation but without full std – you can enable such features when using them in no_std tests [11]. This is an advanced use-case, but worth noting if you want to run certain tests in a no_std mode.

- **Target-specific `cfg`**: In a few cases, you might use `#[cfg(target_os = "none")]` to detect "bare metal" and `#[cfg(not(target_os = "none"))]` for "host" as an alternative to feature flags. This can be useful to provide automatic defaults (for example, use a dummy hardware interface on host vs real one on device without requiring the user to toggle a feature). However, be cautious – it's often clearer to use explicit features because not all “host” targets have an OS (`wasm32-unknown-unknown` is no_std but not `target_os = "none"`). Features give the crate user control to force certain dependencies on or off. In general, **prefer feature gating** for dependencies and code, and use target `cfg` for truly low-level target-specific code (like assembly, architecture-specific instructions, etc.).

By combining these conditional compilation techniques, you can maintain one codebase that builds in multiple configurations. The key is to **keep the default feature set minimal** (embedded-friendly) and opt-in to heavier functionality.

Organizing Tests, Examples, and Benchmarks

To avoid dev dependencies creeping into your embedded builds, adopt organizational patterns that clearly separate code meant for *development* from code meant for *production (firmware)*:

- **Unit tests vs. Integration tests**: You can still write unit tests inside your `#![no_std]` library (using `#[cfg(test)]`) – these will use `std` by default during `cargo test` (Rust’s test harness uses std). This is fine as long as the dev deps required for those tests are no_std-friendly or gated by features. Alternatively, put complex tests in the `mylib-tests` integration crate as discussed. This has the advantage that you can run them with `cargo test -p mylib-tests` on the host without ever enabling `std` or dev features in `mylib-core` itself. As an example, the Embassy project’s `messages` crate (a no_std library) includes QuickCheck tests by adding `quickcheck` as a dev-dependency and using it in `#[cfg(test)]` code [12] [13]. This works because those tests only run on host. If such dev deps become troublesome, moving them out is an option, but for lightweight cases it’s acceptable to keep them as dev-deps (with the caveat that building any example will bring them in). Always ensure any dev dependency used in a no_std crate (for testing) is either optional or can compile under no_std (use `default-features = false` if needed as in the proptest example).

- **Examples**: Structure your examples in a way that **embedded examples do not depend on host-only libraries**. For instance, an example that runs on a microcontroller should use only `no_std` libraries (possibly your crate and a HAL, plus a panic handler). If you need to provide an example that demonstrates the library in a PC simulation (maybe using `std::net` or other facilities), keep it separate as discussed. Many projects put embedded examples in the main crate’s `examples/` for easy reference, and any complex example that needs an OS (networking, large data sets, etc.) in a separate crate or even a different workspace member (to clearly pull in `std` support). By doing so, a command like `cargo build --examples --no-default-features --target thumbv7m-none-eabi` will compile only the device-appropriate examples, and not attempt ones that need `std` (because those live elsewhere or are gated by features).

- **Benchmarks**: Rust’s criterion benchmarks are typically placed in a `benches/` directory of the crate and run with `cargo bench`. However, if the crate is no_std by default, compiling Criterion for it can be a pain (and is unnecessary on embedded). The solution is as above: have a separate bench harness crate. Note that each file in a `benches/` directory is essentially its own crate that depends on your library [14]. By moving `benches/` out of the core crate, you ensure `cargo build` doesn't inadvertently build the benchmark. You can even entirely exclude the bench crate from the workspace by default if you don't want CI to even consider it [10]. Another approach is to use cargo features to only include Criterion when needed, but this gets messy – a dedicated bench crate is simpler and aligns with the principle of separation.

- **Dev dependency versions**: One practical tip – if you keep dev crates out of the main crate, you reduce the risk of version conflicts or feature unifications affecting the library. For example, earlier issues occurred where an embedded-hal crate's dev-dependency activated a `std` feature in a common dependency and broke the build [7]. By isolating dev deps, you decouple their features from the core crate's. And since dev crates are not published, you don't have to worry about them for end users.

Build and CI Automation

Managing a multi-target project can benefit from custom build scripts or task runners to simplify common flows. Two popular approaches in Rust are using a **cargo-xtask** utility crate or a `just` **recipe file**, as well as traditional Makefiles for CI.

- **Cargo xtask**: This is a pattern where you create a small binary crate (often named `xtask`) in your workspace that implements various project-specific commands. For example, you might implement commands like `cargo xtask build-embedded`, `cargo xtask test-all`, `cargo xtask flash` etc. Internally, this crate can use `std::process::Command` to invoke Cargo with the right flags, or even use Rust API calls if suitable. The benefit is you can encapsulate complex logic (building all examples for multiple targets, running host-target integration tests, etc.) in Rust code and make it easy to run via Cargo. As Ferrous Systems puts it, *"cargo-xtask… is a technique for creating custom Cargo subcommands within the scope of a single project – a bit like creating* `make` *rules but in Rust."* [15] In their example, they use `cargo xtask` to orchestrate an entire matrix of tests: one xtask flashes the firmware to a board and runs host-vs-target tests, another runs host-only tests, another runs on-target tests via `probe-run` [16] [17]. This keeps the workflow unified (you can do everything with `cargo xtask ...` commands, which in turn call the necessary `cargo build/test` subcommands with appropriate arguments and `--exclude` flags, etc.). For our purposes, you might have an xtask that ensures the core library builds for all supported targets (running `cargo build -p mylib-core --no-default-features --target X` for a list of architectures), or one that runs all host tests and benches in one go. This can be hooked into CI to avoid duplicating logic in bash scripts.

- **Justfile / Makefile**: If you prefer not to write a Rust harness for tasks, a **Justfile** (for the [`just` command runner]) or a Makefile can be used to define recipes for common operations. For example, you could have a `just build_embed` that calls:

```
cargo build -p mylib-core --no-default-features --target thumbv7m-none-eabi
```

and a `just test_host` that does:

```
cargo test -p mylib-tests
```

and so on. Similarly, a Makefile could orchestrate these steps. The advantage of using such tools is to **prevent human error** and keep CI scripts simple – developers can run a single command that takes care of building or testing in the correct mode, enabling the right features, etc. These external task runners don't integrate as tightly with Cargo as xtask does, but they are straightforward and require no additional compile steps.

Using either approach (or both) can improve maintainability. For example, if certain crates should never be built in certain contexts, your xtask or scripts can enforce that (like using `--exclude` for dev crates when doing an embedded build). In the Ferrous Systems blog, their xtask commands explicitly exclude the host-target test crate when running host tests [17], ensuring the workspace build doesn't accidentally pull in that crate at the wrong time. You can adopt a similar strategy (e.g., exclude bench crates from a normal `cargo test --workspace` run).

Nightly Options and Trade-offs

The good news is that **all of the above can be accomplished on stable Rust/Cargo**. We avoided any nightly-only features. In edge cases where you absolutely cannot restructure and need to prevent Cargo from even considering dev deps, there are a couple of nightly flags (like `-Z avoid-dev-deps`) that *"prevent the resolver from including dev-dependencies during resolution."* [18] However, these are typically not necessary if you organize your workspace as recommended. Another nightly Cargo feature is scrubbing examples from docs without dev deps [19], but again, that's a niche scenario.

If you find yourself pushing against a limitation of stable Cargo (for example, maybe you want to conditionally include a dev dependency only when a certain feature is enabled – currently dev-deps don't support `optional = true`), you might consider an RFC or a build script workaround. In general, though, sticking to stable patterns (like target-specific dev sections, optional normal dependencies, and separate crates) yields a cleaner and more future-proof solution than any hack that relies on nightly.

Trade-off summary: Splitting crates and using features may add a bit of upfront complexity (more Cargo.toml files, more coordination), but it pays off with **faster build times, clearer boundaries, and fewer cfg headaches** in the code. Each crate can be tested and built in isolation. On the other hand, keeping everything in one crate with heavy use of `cfg` flags could become hard to maintain, and you risk accidental cross-contamination as the project grows.

Conclusion

By following these best practices, you can create a multi-target Rust project that cleanly supports embedded `no_std` targets and standard-hosted use cases (for testing and tooling). In summary:

- **Use a workspace with multiple crates** to separate core no_std logic from any code that requires `std` or dev-only libraries.
- **Keep dev-dependencies out of the core crate** – use separate test/bench crates or target-specific Cargo sections so they aren't compiled for device builds.

- **Leverage Cargo features (std/alloc)** to make your library dual-mode, and use `cfg(feature = "...")` to isolate host-specific code.
- **Organize examples and tests thoughtfully**, so embedded examples remain lightweight and host-centric examples/tests live elsewhere.
- **Automate common build tasks** with xtask or just, to enforce the intended build flows and simplify CI.

Following patterns used by major embedded projects (like Embassy's layered approach and Ferrous Systems' host/target split) will result in a maintainable project structure [9]. This ensures that adding a new benchmark or host test in the future won't inadvertently break your `thumbv7m-none-eabi` build. It also makes it easier for contributors to understand which code is meant for the embedded target vs. the host. With a clean separation, your CI can reliably test the core crate in a bare-metal context (to guarantee no `std` sneak ins) and run rich tests on the host, giving you the best of both worlds in an idiomatic, stable Rust setup.

Sources:

- Cargo Reference – Dev-dependencies and target-specific dependencies [20] [2]
- User discussions on `no_std` workspace dev-deps (Rust Users Forum) [2] [3]
- Rust Edition Guide – New feature resolver behavior [6]
- Ferrous Systems Embedded Testing blog – project structure and xtask usage [9] [15]
- Rust Cargo Book – Example of cargo-xtask in use [17] and general dev-dependency guidance.

---

[1] [4] [5] [20] Specifying Dependencies - The Cargo Book
https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html

[2] [3] Compile example sketch for Feather M0 board - embedded - The Rust Programming Language Forum
https://users.rust-lang.org/t/compile-example-sketch-for-feather-m0-board/124632

[6] Default Cargo feature resolver - The Rust Edition Guide
https://doc.rust-lang.org/edition-guide/rust-2021/default-cargo-resolver.html

[7] [8] Dev-dependencies force features in dependencies so that cargo build is not possible - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/dev-dependencies-force-features-in-dependencies-so-that-cargo-build-is-not-possible/21221

[9] [12] [13] [15] [16] [17] Testing an embedded application - Ferrous Systems
https://ferrous-systems.com/blog/test-embedded-app/

[10] Don't build criterion in CI · Issue #106 · ruma/ruma - GitHub
https://github.com/ruma/ruma/pull/106

[11] Support `no_std` Usage · Issue #47 · proptest-rs/proptest · GitHub
https://github.com/proptest-rs/proptest/issues/47

[14] How to build a Custom Benchmarking Harness in Rust - Bencher
https://bencher.dev/learn/benchmarking/rust/custom-harness/

[18] [19] Unstable Features - The Cargo Book
https://rustwiki.org/en/cargo/reference/unstable.html