A Layered Compendium of Idiomatic Rust Patterns: From Application to Bare Metal

Introduction

The Philosophy of Idiomatic Rust

The Rust programming language is distinguished not merely by its feature set but by a cohesive design philosophy that profoundly influences its idiomatic usage. The core tenets of memory safety without a garbage collector, zero-cost abstractions, and fearless concurrency are not isolated features; they are the architectural pillars upon which the language's ecosystem is built.[1] Understanding idiomatic Rust, therefore, requires more than memorizing syntax or library functions. It necessitates an appreciation for how these principles shape the solutions to common programming problems, from high-level application logic to low-level hardware manipulation. Effective Rust code is not just code that compiles; it is code that leverages the type system and ownership model to provide compile-time guarantees about its correctness, safety, and performance.

This report undertakes a systematic exploration of idiomatic Rust by dissecting the language's application across distinct environmental layers. This layered methodology provides a framework for understanding how Rust's core principles are applied under varying constraints. By examining the patterns that emerge in different contexts—from the resource-rich environment of a modern operating system to the highly constrained world of bare-metal microcontrollers—a deeper, more nuanced understanding of the language's design and its practical implications can be achieved.

Structure of the Report

The analysis is structured into three principal parts, each corresponding to a progressively more constrained development environment. This structure follows a gradient that reveals how Rust's idioms adapt, evolve, and sometimes transform to meet the challenges of their respective domains.

- **Part I: Foundations in the Standard Library (std)** examines the patterns prevalent in application development where the full Rust Standard Library (std) and an underlying operating system are available. These idioms form the bedrock of most Rust programming.
- **Part II: Building for Constrained Environments (no_std)** shifts the focus to contexts where the std library is partially or entirely absent, such as in embedded systems, WebAssembly modules, and certain operating system components.
- **Part III: Systems Programming at the Bare Metal** delves into the deepest layer of systems programming, exploring the patterns used in the development of operating system kernels and kernel modules, where direct hardware interaction and the management of unsafe code are paramount.

Through this layered exploration, a comprehensive compendium of idiomatic patterns will be assembled, providing a robust guide for developers seeking to write efficient, maintainable, and high-quality Rust code across the entire systems programming spectrum.

Part I: Foundations in the Standard Library (std)

Development with the Rust Standard Library (std) represents the most common use case for the language, encompassing everything from command-line utilities to sophisticated, high-performance web services. In this environment, the developer has access to a rich set of pre-built abstractions for interacting with the underlying operating system. The std library provides modules for fundamental operations such as I/O, networking, threading, and dynamic collections.[3]

These modules are not merely convenience wrappers; they constitute a carefully designed, safe, and cross-platform API that abstracts away the complexities and inconsistencies of various OS-specific functionalities. Operating systems typically expose their primitives—for managing threads, files, sockets, and more—through unsafe, platform-dependent C application binary interfaces (ABIs). A primary function of the std library is to provide a unified and safe interface over these disparate primitives.[3] For example,

std::thread offers a consistent, safe API that internally utilizes pthreads on POSIX-compliant systems and the CreateThread API on Windows.

Crucially, Rust's core principles of ownership and type safety are deeply integrated into these std abstractions to enforce correctness at compile time. The std::fs::File type, for instance, implements the Drop trait to guarantee that file handles are automatically closed when they go out of scope. This design prevents a common class of resource leak bugs that frequently occur with manual resource management paradigms like fopen/fclose in C. Consequently, the std library itself can be viewed as a foundational pattern: the **safe abstraction boundary**. It encapsulates the inherently unsafe interactions with the host operating system, enabling the vast majority of application code to be written entirely within Rust's robust safety guarantees. For developers working within the std environment, the idiomatic approach is to fully leverage these high-level, safe abstractions rather than attempting to bypass them. The patterns detailed in this section are centered on the effective and efficient use of these foundational tools.

Pattern Suite 1.1: Expressive and Resilient Error Handling

This suite of patterns addresses the canonical approach to managing operational success and failure in Rust applications. The emphasis is on leveraging the type system to achieve compile-time correctness, eliminate entire classes of runtime errors, and clearly communicate the fallibility of operations.

The Result and Option Dichotomy: Signifying Possibility vs. Failure

The distinction between Option<T> and Result<T, E> is a cornerstone of idiomatic Rust error handling. It provides a type-safe, explicit mechanism for dealing with two fundamentally different scenarios: the potential absence of a value versus the potential failure of an operation.[8]

- **Core Pattern:** Option<T> is used to represent a value that may or may not be present. It is an enum with two variants: Some(T), which contains a value, and None, which signifies its absence. This is the idiomatic choice for functions where "not found" or "no value" is an expected and valid outcome. For example, searching for an element in a collection might return None if the element is not found; this is not an error, but a normal result of the operation.
- **Core Pattern:** Result<T, E> is used for operations that can fail. It is an enum with two

variants: Ok(T), containing the successful result value, and Err(E), containing a value that describes the nature of the error. This is the idiomatic choice for operations like file I/O, network requests, or parsing, where external factors or invalid input can cause the operation to fail.

This clear separation in the type system forces developers to confront both possibilities at compile time. The compiler ensures that all variants of an Option or Result are handled, typically through a match expression or by using one of the many combinator methods provided for these types. This design makes it impossible to accidentally use a null value, as the concept does not exist in safe Rust, thereby preventing null pointer exceptions. Furthermore, it prevents unhandled exceptions from crashing a program at runtime, as the possibility of failure is encoded in the function's return type and must be addressed by the caller.[10] This approach promotes composability; unlike panicking, which unwinds the stack and terminates the thread, returning a

Result allows the caller to decide how to handle the failure, enabling robust and resilient software design.[9]

Idiomatic Propagation with the ? Operator

While match expressions provide exhaustive and explicit handling of Result and Option types, they can lead to verbose and nested code, particularly in functions that perform multiple fallible operations. The ? operator is a syntactic sugar that provides an ergonomic and idiomatic pattern for error propagation.

- **Core Pattern:** Within a function that itself returns a Result or Option, the ? operator can be appended to an expression that evaluates to a Result or Option. If the value is Ok(T) or Some(T), the operator unwraps the value to T. If the value is Err(E) or None, the operator immediately returns from the current function, propagating the Err(E) or None value to the caller.[8]

The following example demonstrates the transformation from manual matching to using the ? operator for a function that reads the contents of a file into a string:

Rust

```
// Manual matching
use std::fs::File;
```

```rust
use std::io::{self, Read};

fn read_file_contents_manual(path: &str) -> Result<String, io::Error> {
    let file_result = File::open(path);
    let mut file = match file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut contents = String::new();
    match file.read_to_string(&mut contents) {
        Ok(_) => Ok(contents),
        Err(e) => Err(e),
    }
}

// Idiomatic propagation with `?`
fn read_file_contents_idiomatic(path: &str) -> Result<String, io::Error> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

The ? operator is more than a mere shorthand; it fundamentally improves the ergonomics of writing fallible code. It allows functions that handle errors to be written with nearly the same clarity and conciseness as functions that do not, encouraging developers to use Result pervasively. This transforms error handling from a syntactically burdensome task into a clean and integrated aspect of the language's control flow.

Designing Custom Error Types with thiserror

For libraries and large applications, returning a generic error type like String or Box<dyn std::error::Error> is often insufficient. A String does not implement the standard std::error::Error trait, making it incompatible with many error-handling libraries, and it discards valuable type information about the source of the error.[12] A more robust pattern is to define a custom error

enum that represents all possible failure modes within a specific domain.

- **Core Pattern:** Define a custom enum where each variant represents a distinct category

of error. Use the thiserror crate to automatically derive the necessary traits (std::error::Error, std::fmt::Display, and std::convert::From), which significantly reduces boilerplate code.[12]

This pattern allows consumers of a library to programmatically inspect the error and implement different logic for different failure conditions. For example, a caller might want to retry an operation that failed due to a transient network error but immediately fail if the error was due to invalid user input.

Rust

```
use thiserror::Error;
use std::io;
use std::num::ParseIntError;

#
pub enum DataProcessingError {
    #[error("Failed to read data from source")]
    Io(#[from] io::Error),

    #[error("Failed to parse input data")]
    Parse(#[from] ParseIntError),

    #[error("Invalid input value: {0}")]
    InvalidValue(i32),
}

fn process_data() -> Result<(), DataProcessingError> {
    let data = std::fs::read_to_string("data.txt")?; // io::Error converted via #[from]
    let number: i32 = data.trim().parse()?; // ParseIntError converted via #[from]

    if number < 0 {
        return Err(DataProcessingError::InvalidValue(number));
    }

    //... process number...
    Ok(())
}
```

In this example, thiserror's #[derive(Error)] macro implements the Error and Display traits. The

#[error("...")] attribute defines the display format, and #[from] automatically generates From trait implementations, allowing the ? operator to transparently convert underlying errors (like io::Error) into the custom DataProcessingError type.

Application-Level Error Reporting with anyhow

While libraries benefit from specific, well-defined error types, the top level of an application (such as the main function of a binary or a web request handler) often has a different requirement. At this boundary, the primary goal is typically to report the error to the user or a logging system in a clear, context-rich manner, rather than to programmatically handle different error variants. For this use case, the anyhow crate provides a highly ergonomic pattern.

- **Core Pattern:** In application code, use anyhow::Result<T> as the return type for functions. anyhow::Error is a type-erased error type (essentially a wrapper around Box<dyn std::error::Error>) that can hold any error type that implements std::error::Error. The context() method can be used to add descriptive, human-readable context to errors as they are propagated up the call stack.[12]

Rust

```
use anyhow::{Context, Result};

fn main() -> Result<()> {
    let config = load_config("app.toml")
        .context("Failed to load application configuration")?;

    run_app(config)
        .context("Application execution failed")?;

    Ok(())
}

fn load_config(path: &str) -> Result<String> {
    //... implementation that returns Result<String, E>...
    # Ok("config".to_string())
}
```

```
fn run_app(config: String) -> Result<()> {
    //... implementation...
    # Ok(())
}
```

This pattern elegantly resolves the trade-off between specificity and convenience. Libraries should use specific error types with thiserror to provide maximum information to their consumers. Applications can then use anyhow to consume these diverse error types, wrap them in a uniform anyhow::Error type, and add contextual information that is meaningful at the application level. This creates a clear and effective separation of concerns in error handling strategy.

Pattern Suite 1.2: Efficient Data Structures with std::collections

Rust's standard library offers a rich set of collection types, each with specific performance characteristics and memory layouts. Idiomatic use of these collections involves not only choosing the correct type for the task but also leveraging their APIs to maximize performance and memory efficiency.

Vec<T>: Capacity Management and Performance Idioms

Vec<T>, a contiguous, growable array type, is one of the most commonly used collections in Rust. Its performance is heavily influenced by how its capacity is managed.

- **Core Pattern:** When the final size of a Vec is known or can be reasonably estimated beforehand, pre-allocate its capacity using Vec::with_capacity(n). This pattern avoids the performance overhead of repeated reallocations.[14]

A Vec has both a length (the number of elements it currently contains) and a capacity (the number of elements it can hold without reallocating). When an element is pushed to a Vec that is at full capacity, the vector must perform a new allocation on the heap for a larger buffer (typically doubling in size), copy all of its existing elements from the old buffer to the new one, and then deallocate the old buffer. This is a computationally expensive operation, involving memory allocation and a memcpy. By pre-allocating with with_capacity, these intermediate reallocations can be entirely avoided, leading to significant performance gains, especially when building large vectors in a loop. This pattern exemplifies Rust's principle of

giving developers control over low-level details when performance is critical.

HashMap<K, V>: Effective Use of the Entry API

HashMap<K, V> provides average-case O(1) performance for insertions, lookups, and removals. A common task is to insert a value only if a key is not present, or to modify an existing value in place. The entry API provides an efficient and idiomatic way to perform these operations.

- **Core Pattern:** To conditionally insert or modify an entry in a HashMap, use the entry() method. This method returns an Entry enum, which represents a potential entry in the map. Its or_insert() and or_insert_with() methods allow for the insertion of a value only if the entry is vacant, all within a single lookup.[14]

Consider the task of counting word frequencies:

Rust

```
use std::collections::HashMap;

let text = "hello world hello";
let mut counts = HashMap::new();

// Non-idiomatic approach with two lookups
for word in text.split_whitespace() {
    let count = counts.get_mut(word);
    match count {
        Some(c) => *c += 1,
        None => {
            counts.insert(word, 1);
        }
    }
}

// Idiomatic approach with the entry API
let mut counts = HashMap::new();
for word in text.split_whitespace() {
    let count = counts.entry(word).or_insert(0);
```

```
    *count += 1;
}
```

The non-idiomatic version may perform two lookups for each new word: one for get_mut and another for insert. The entry API, in contrast, performs only a single lookup. It calculates the hash of the key once and finds the corresponding bucket. If the entry is vacant, it allows a value to be inserted directly; if it is occupied, it provides a mutable reference to the existing value. This pattern is not only more performant but also more concise and expressive.

Choosing the Right Collection: Performance and Ergonomic Trade-offs

The standard library provides several collection types, and the idiomatic choice depends on the specific requirements of the algorithm, particularly its data access patterns.

- **Core Pattern:** For general-purpose use, Vec<T> and HashMap<K, V> are the default choices for sequence and map-like data, respectively. They offer the best all-around performance for the most common operations.[14]
    - **Use Vec<T>** when you need O(1) indexed access, efficient iteration, and amortized O(1) push/pop at the end.
    - **Use HashMap<K, V>** when you need average-case O(1) key-based lookups, insertions, and removals, and the order of elements is not important.
- **Alternative Patterns:**
    - **Use BTreeMap<K, V>** when you require key-based lookups and need to iterate over the elements in sorted key order. Its operations are O(log n), making it slower than HashMap for individual lookups but providing ordered traversal.
    - **Use VecDeque<T>** when you need a double-ended queue with efficient O(1) push and pop operations at both the front and the back of the sequence.
    - **Use HashSet<T> or BTreeSet<T>** when you only need to store unique values and perform efficient membership tests.

The choice of a collection is a declaration of algorithmic intent. Selecting Vec signals a reliance on contiguous memory and indexed access, while selecting HashMap signals a need for fast key-value lookups. In some performance-critical scenarios, it can even be beneficial to use a combination of structures. For instance, a common pattern involves building an initial mapping from a unique identifier to an index using a HashMap, and then storing the actual data in a Vec for fast, cache-friendly indexed access and iteration once the initial mapping is complete.[16] This demonstrates a sophisticated understanding of how the underlying memory layout and performance characteristics of each collection can be leveraged for optimal results.

Pattern Suite 1.3: Fearless Concurrency

Rust's approach to concurrency is one of its most celebrated features, famously termed "fearless concurrency." This is not an empty slogan; it is a direct consequence of the ownership and type systems being applied to eliminate data races—one of the most insidious classes of bugs in concurrent programming—at compile time.[2]

Shared-State Concurrency: The Arc<Mutex<T>> Pattern Explained

When multiple threads need to access and modify the same piece of data, Rust's core safety rules present a challenge. The ownership system prevents a value from having multiple owners, and the borrowing rules prevent the existence of multiple mutable references (&mut T) to the same data. The idiomatic solution to this is the Arc<Mutex<T>> pattern.[17]

- **Core Pattern:** To safely share mutable state across threads, the data is wrapped in two layers of smart pointers:
    1. std::sync::Mutex<T>: A Mutex (mutual exclusion) provides interior mutability. It guards the data T, ensuring that only one thread can acquire a lock and gain mutable access at any given time. The lock is managed by a MutexGuard smart pointer, which automatically releases the lock when it goes out of scope (via the Drop trait), preventing deadlocks from forgotten unlocks.
    2. std::sync::Arc<T>: An Arc (Atomically Reference Counted) is a thread-safe smart pointer that allows multiple threads to have shared ownership of a value. It is the thread-safe equivalent of Rc<T>. When Arc::clone() is called, it increments an atomic reference counter, providing a new Arc handle to the same underlying data without violating ownership rules.

The combination, Arc<Mutex<T>>, solves both problems simultaneously. Arc handles the shared ownership, allowing multiple threads to hold a reference to the Mutex. The Mutex then handles the exclusive access, ensuring that only one thread at a time can mutate the data within.

This pattern is the canonical embodiment of Rust's approach to safe shared-state concurrency. It transforms a complex runtime problem (preventing data races) into a type-system construct that the compiler can statically verify. While the type signature Arc<Mutex<T>> may appear verbose, it is an explicit and provably safe declaration of intent.[22]

Message-Passing Concurrency with Channels (mpsc)

An alternative to shared-state concurrency is the message-passing model, which is often easier to reason about as it avoids the complexities of locking. Rust's standard library provides channels for this purpose in the std::sync::mpsc module (multiple producer, single consumer).

- **Core Pattern:** Threads communicate by sending messages through a channel. A channel consists of a transmitter (Sender) and a receiver (Receiver). The Sender can be cloned to allow multiple threads to send messages, while the Receiver waits for incoming messages.

This pattern aligns with the design philosophy of "sharing memory by communicating." When a value is sent through a channel, its ownership is transferred to the receiving thread. The Rust compiler enforces this transfer, making it impossible for the sending thread to accidentally access the value after it has been sent. This use of the ownership system provides thread safety without requiring explicit locks, often leading to more decoupled and easier-to-understand concurrent architectures.

Choosing a Concurrency Model: Trade-offs and Best Practices

The choice between shared-state and message-passing concurrency is an architectural decision that depends on the problem being solved.

- **Guideline:**
  - **Prefer message passing** when the concurrent tasks are loosely coupled and can be modeled as independent actors processing streams of data. This often leads to simpler and more modular designs.
  - **Use shared state** when multiple threads require frequent, fine-grained access to a large, complex data structure. In such cases, repeatedly sending the entire structure or fragments of it over a channel would be inefficient compared to modifying it in place under a lock.

While Rust's compiler prevents data races, it cannot prevent logical errors like deadlocks. A deadlock can occur in a shared-state model if, for example, Thread A holds a lock on resource X and waits for resource Y, while Thread B holds the lock on Y and waits for X. This is a design-level problem that requires careful lock ordering and architectural planning. The explicitness of the Arc<Mutex<T>> pattern helps to make these potential contention points

visible, but it does not eliminate the need for careful design.

Pattern Suite 1.4: High-Performance I/O and Asynchronous Programming

This suite covers idiomatic patterns for efficient data processing and I/O, spanning from traditional synchronous operations to the modern asynchronous paradigm that powers high-performance network services.

Leveraging Read, Write, and Buffered I/O for Efficiency

For synchronous I/O operations, performance is often dictated by the number of interactions with the operating system. Each system call (e.g., a read or write syscall) involves a context switch, which is a relatively expensive operation.

- **Core Pattern:** To minimize system calls and improve throughput, wrap I/O-bound types like std::fs::File or std::net::TcpStream in the buffered wrappers std::io::BufReader or std::io::BufWriter. These wrappers maintain an in-memory buffer, allowing many small read or write operations to be consolidated into a single, larger system call.[23]

This is an application of the decorator pattern, where the BufReader adds buffering functionality to any type that implements the Read trait, without changing the underlying reader's interface. The Read and Write traits themselves are a powerful abstraction, providing a generic interface for any source or sink of bytes. This allows for the creation of composable I/O pipelines.

The Zero-Cost Abstraction: Advanced Iterator Patterns

Rust's iterators are a prime example of its zero-cost abstraction philosophy. They provide a high-level, declarative API for data processing that compiles down to highly efficient, low-level machine code.

- **Core Pattern:** Prefer using iterator chains—a series of method calls like .iter().map(...).filter(...).sum()—over manually written for loops. The compiler is exceptionally good at optimizing these chains, often unrolling loops and inlining closures

to produce code that is as fast or faster than the hand-written equivalent.[25]

The power of this pattern stems from the lazy nature of iterators.[26] Iterator adapters like

map and filter do not perform any work when they are called. Instead, they construct a new iterator object that encapsulates the desired operation. No processing occurs until a consuming method, such as collect(), sum(), or for_each(), is called. This laziness allows the compiler to view the entire sequence of operations as a single unit and apply powerful optimizations like loop fusion, which merges multiple loops into one, eliminating intermediate collections and improving cache locality. This enables developers to write expressive, high-level code without sacrificing performance.

Asynchronous Programming with Tokio: Core Patterns

For I/O-bound applications, such as network servers that must handle thousands of concurrent connections, the traditional model of one thread per connection is not scalable. Asynchronous programming, powered by runtimes like Tokio, is the idiomatic solution in Rust.

- **Core Pattern:** Structure the application around the async and .await keywords. async fn defines an asynchronous function that returns a Future. This Future is a state machine that can be polled by an async runtime, like Tokio. When an async function reaches an .await point on an I/O operation, it yields control back to the runtime, allowing another task to run on the same thread. This enables a single thread to manage a vast number of concurrent I/O operations efficiently.[29]
- **Shared State in Tokio:** The Arc<Mutex<T>> pattern is also applicable in asynchronous code. However, a critical distinction must be made:
  - If a lock is held across an .await point, the Mutex from tokio::sync **must** be used. The std::sync::Mutex is a blocking mutex; if a task holding its lock were to .await, it would block the entire worker thread, preventing other tasks from making progress. tokio::sync::Mutex is an async-aware mutex; its .lock().await method will yield to the scheduler if the lock is contended, rather than blocking the thread.
  - For very short critical sections that do not involve .await, std::sync::Mutex is often preferred due to its lower performance overhead.[31]
- **Error Handling in Axum:** Web frameworks built on Tokio, such as Axum, integrate Rust's error handling philosophy into their design.
  - **Core Pattern:** Handler functions in Axum typically return a Result<T, E>, where the success type T implements the IntoResponse trait, and the error type E also implements IntoResponse. This allows the ? operator to be used for error propagation within the handler. If an error is returned, Axum automatically calls .into_response() on it to generate an appropriate HTTP error response (e.g., a 500

Internal Server Error).[33] This creates a seamless and robust error handling workflow for web applications.

- **State Management in Axum:** Sharing application-wide state, such as a database connection pool, is a common requirement.
  - **Core Pattern:** The application state is typically wrapped in an Arc to allow for shared ownership. This Arc<AppState> is then passed to the Axum Router using the .with_state() method. Individual handlers can then access this shared state in a type-safe manner by using the State extractor as a function argument: async fn handler(State(state): State<Arc<AppState>>).[36] This pattern provides a clean, type-safe mechanism for dependency injection in Axum applications.

Part II: Building for Constrained Environments (no_std)

This section transitions from the resource-rich context of std-enabled applications to the constrained environments typical of embedded systems, WebAssembly, and low-level OS components. In these domains, the full standard library is unavailable, forcing a more deliberate and controlled approach to programming. The no_std attribute (#![no_std]) signals to the compiler that the crate will not link against std, instead relying on a smaller, platform-agnostic subset of the library.

The concept of no_std development in Rust is not a binary choice but rather a spectrum of capabilities, defined by a layered hierarchy within the standard library itself.[39] At the foundation lies the

core crate. It is the absolute minimal, platform-agnostic subset of std, containing fundamental language primitives like Option, Result, iterators, and atomic types. core makes zero assumptions about the target environment, critically, not even the presence of a heap for dynamic memory allocation.[42]

Building upon core is the alloc crate. This crate introduces heap-allocated data structures that are familiar from std, such as Box, Vec, and String. However, unlike std, alloc does not provide a memory allocator itself. Instead, it requires the developer to supply an implementation of the GlobalAlloc trait, giving explicit control over how heap memory is managed.[40]

Finally, the std crate builds on both core and alloc, adding features that depend on an underlying operating system, such as threading, networking, file I/O, and a default system allocator.[39] This layered architecture is a cornerstone of Rust's portability. It allows the language to target a vast range of devices by enabling developers to select only the layers of

abstraction that their environment can support. Consequently, idiomatic

no_std programming is an exercise in conscious architectural choices, selecting the appropriate layer and writing code that is either tailored to or generic over these capabilities.

Feature	core	alloc	std
Language Primitives (Option, Result, Iterator)	✓	✓	✓
Slices, str, Primitive Types	✓	✓	✓
Atomic Operations	✓	✓	✓
Heap Allocation (Box, Vec, String)	✘	✓*	✓
Global Memory Allocator	✘	✘ **	✓
Threading and Synchronization (Mutex)	✘	✘	✓
Networking (TCP/UDP)	✘	✘	✓
File System I/O	✘	✘	✓
Standard Runtime (e.g., main setup)	✘	✘	✓
** Available, but requires a user-provided global allocator.*			

*** Does not provide an allocator, but requires one to be defined.*			

Pattern Suite 2.1: The core, alloc, and std Hierarchy

This suite presents patterns for writing portable Rust code that can operate across the no_std spectrum, from core-only libraries to those that can optionally leverage std.

Idiomatic Library Design: Creating Conditionally no_std Crates

To maximize reusability, library crates should be designed to support the most constrained environment possible. A library that performs data serialization, for example, may not need file I/O or threading and can be made no_std compatible, allowing it to be used in embedded systems and WebAssembly in addition to standard applications.

- **Core Pattern:** Design libraries to be no_std by default. Use a Cargo feature, conventionally named std, to conditionally compile code that requires the standard library. This allows consumers of the library to opt-in to std-dependent functionality when it is available.[40]

There are two common techniques to implement this pattern, with one being strongly preferred for its consistency. The first, and less ideal, approach is:

Rust

```
// Less idiomatic: changes the prelude
#![cfg_attr(not(feature = "std"), no_std)]
```

While this works, it has a subtle side effect: the implicit prelude of items brought into scope changes depending on whether the std feature is enabled. With the feature, the std prelude is used; without it, the core prelude is used. This can lead to confusing compilation errors and

inconsistencies in use statements across the codebase.[45]

The superior and more idiomatic pattern is to declare the crate as unconditionally no_std and then conditionally bring in the std crate when the feature is enabled:

Rust

```
// Idiomatic: always `no_std`, consistent prelude
#![no_std]

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

This approach ensures that the crate always uses the core prelude, providing a consistent development experience regardless of the feature flags. Any use of std functionality must be explicitly prefixed with std::, making the code clearer and easier to reason about. This pattern is strongly recommended for creating portable, optionally-std libraries.[45]

- **CI Verification:** The Rust compiler will not error if a no_std crate has a dependency that uses std. This can inadvertently break the no_std compatibility of a library. The only robust way to prevent this is through continuous integration (CI).
    - **Core Pattern:** Implement a CI job that attempts to build the crate for a known no_std target, such as thumbv7em-none-eabi (a common ARM Cortex-M target), with default features disabled (--no-default-features). Any dependency that transitively requires std will cause this build to fail, providing an immediate and reliable check of no_std compatibility.[40]

Pattern Suite 2.2: Deterministic Memory Management

In constrained environments, memory is a precious resource, and dynamic allocation can be a source of non-determinism and failure. Idiomatic no_std patterns for memory management prioritize predictability and explicit handling of allocation failures.

Strategy	Performance	Determinism	Error Handling	Memory Overhead	Flexibility

Stack Allocation	Very High (O(1))	High (compile-time size)	Compile-time size checks	Low	Low (fixed size)
heapless Crate	High (stack-based)	High (fixed capacity)	Runtime Result on overflow	Low	Medium (dynamic length)
alloc with Global Allocator	Varies by allocator	Varies (can be non-deterministic)	panic on OOM (by default)	Medium (heap overhead)	High (dynamic size)
Custom Allocators	Tunable	Tunable (e.g., bump is deterministic)	User-defined	Tunable	High

Stack-First Design and the heapless Crate

In the most constrained, core-only environments, there is no heap. All memory must be allocated either in the static data section of the binary or on the call stack. While this provides extreme predictability, it limits the use of dynamically sized data structures.

- **Core Pattern:** For data structures that need to grow and shrink at runtime without a heap, use the heapless crate. This crate provides implementations of Vec, String, HashMap, and other collections that are backed by a fixed-size array allocated on the stack or in a static variable.[48]

The key feature of heapless collections is their approach to error handling. Operations that could cause the collection to exceed its fixed capacity, such as Vec::push, do not panic. Instead, they return a Result, with Ok on success and Err containing the element that could not be pushed on failure. This forces the developer to explicitly handle the "out of memory" condition at every call site, which is essential for building robust, high-reliability embedded systems where unexpected panics are unacceptable.[49]

Patterns for Using the alloc Crate

When a heap is available but the full std library is not (e.g., in an OS kernel or some embedded systems), the alloc crate can be used.

- **Core Pattern:** To enable alloc, add extern crate alloc; to the crate root and provide an implementation of the GlobalAlloc trait. This makes heap-allocated types like alloc::vec::Vec available for use.[40]

A significant challenge with the standard alloc crate is its default assumption of infallible allocation. Most allocation-related methods will panic if the allocator returns an out-of-memory error. This behavior is often undesirable in constrained environments where allocation failures are a real possibility that must be handled gracefully.

- **Fallible Allocation Pattern:** An emerging and increasingly important pattern is to use the try_ variants of allocation methods, such as Box::try_new and Vec::try_reserve. These methods return a Result instead of panicking on allocation failure, allowing the program to handle the error. While support for fallible allocation is not yet pervasive across the entire collections API, its use is a critical pattern for writing robust code in memory-constrained alloc-enabled environments.[40]

Implementing and Using Custom Allocators

To use the alloc crate in a no_std environment, the developer must provide a global memory allocator.

- **Core Pattern:** Define a static type that implements the unsafe GlobalAlloc trait. This trait requires two methods: alloc and dealloc. This implementation is then registered with the compiler using the #[global_allocator] attribute.[51]

This pattern grants the developer complete control over the system's heap management strategy. Different allocator designs can be chosen to meet specific application requirements:

- **Bump Allocator:** A simple and extremely fast allocator that allocates memory from a contiguous block. Allocations are O(1), but individual deallocations are not supported; the entire memory region is typically freed at once. This is ideal for tasks with distinct phases where all objects created during a phase can be discarded together.[52]
- **Linked-List Allocator:** A more flexible design that maintains a list of free memory blocks. It supports individual deallocations but can be slower and susceptible to memory fragmentation. Crates like linked_list_allocator provide pre-built implementations suitable

for many bare-metal systems.[51]

Pattern Suite 2.3: Hardware Abstraction and Drivers

Interacting with hardware is a central task in embedded programming. Idiomatic Rust patterns for this domain focus on creating safe, portable, and reusable abstractions over low-level hardware details.

The embedded-hal Philosophy: Traits over Implementations

A key challenge in embedded development is the lack of portability. A driver written for a specific microcontroller's peripherals (e.g., its I2C controller) cannot be easily used on a different microcontroller. The embedded-hal project solves this problem through a set of traits that define abstract interfaces for common hardware peripherals.

- **Core Pattern:** Device drivers should be written to be generic over the traits defined in embedded-hal, such as I2c, Spi, and OutputPin. The driver's constructor takes an instance of a type that implements the required trait(s). This decouples the driver logic from any specific hardware implementation.[54]

This is a powerful application of the dependency inversion principle. Instead of the high-level driver depending on low-level hardware details, both depend on the embedded-hal abstractions. The application code is responsible for creating a concrete instance of the hardware peripheral from a specific Hardware Abstraction Layer (HAL) crate (e.g., stm32f4xx-hal or esp-hal) and passing it to the driver. This pattern has fostered a rich ecosystem of platform-agnostic drivers in Rust, significantly accelerating embedded development by promoting code reuse.[58]

Typestate Programming for Peripheral Management

Peripherals often operate as state machines. For example, a GPIO pin can be configured as an input, an output, or an alternate function. Certain operations are only valid in specific states.

- **Core Pattern:** Use Rust's type system to encode the state of a peripheral. A peripheral is represented by a struct with a generic type parameter that signifies its current state.

Functions that change the peripheral's state consume the old struct and return a new one with an updated type state. This ensures that only methods applicable to the current state are available at compile time.[43]

For example, a GPIO pin might be represented as Pin<Input<Floating>>. Calling a .into_push_pull_output() method would consume this pin and return a Pin<Output<PushPull>>. The compiler would then allow .set_high() to be called on the new type, but it would have been a compile-time error to call it on the original input pin. This "typestate" pattern leverages Rust's zero-cost abstractions to create exceptionally safe and expressive hardware APIs, preventing entire classes of logical errors at compile time.

The Singleton Pattern for Accessing Peripherals

In a bare-metal system, there is only one of each physical peripheral (e.g., one I2C1 controller, one GPIO Port A). It is critical to ensure that the program does not create multiple independent software handles to the same physical hardware, as this could lead to conflicting register configurations.

- **Core Pattern:** Ensure that peripheral access structs can only be instantiated once. This is typically achieved by providing a take() method on a Peripherals struct that returns an Option. The first time it is called, it moves the peripherals out of a static context and returns Some(peripherals). All subsequent calls will return None.[60]

This pattern uses Rust's ownership system to enforce at compile time that there is only a single owner for the hardware resources. This statically prevents race conditions and resource conflicts that are a common source of bugs in traditional embedded C programming, where peripheral registers are often accessed through globally accessible volatile pointers.

Pattern Suite 2.4: Concurrency without an OS

On bare-metal systems without an operating system, concurrency is not managed through threads but primarily through hardware interrupts. An interrupt can occur at any time, preempting the main application loop. This creates the potential for race conditions if the interrupt handler and the main loop access the same data.

Managing Interrupts: Critical Sections and Atomics

- **Core Pattern:** To safely share data between the main execution context and an interrupt handler, access to the shared data must be synchronized.
    1. **Critical Sections:** For complex data modifications, use a critical section. This involves temporarily disabling all interrupts, performing the data modification, and then re-enabling interrupts. The cortex_m::interrupt::free function provides a safe abstraction for this on ARM Cortex-M cores. This is a simple but blunt instrument, as it increases interrupt latency for the entire system.[61]
    2. **Atomics:** For simple data types like flags or counters, use the atomic types from core::sync::atomic (e.g., AtomicBool, AtomicU32). These types provide methods that are guaranteed by the hardware to be indivisible (atomic). For example, fetch_add can increment a counter without risk of being interrupted mid-operation, avoiding the need to disable interrupts entirely.[61]

The choice between these patterns involves a trade-off. Critical sections are easy to use for arbitrary data but can negatively impact the real-time performance of a system. Atomics are highly efficient and have minimal impact on latency but are limited to primitive types and operations.

Lightweight Asynchronous Frameworks: Patterns with Embassy

For more complex concurrent applications on bare metal, managing state and interactions through interrupts and global flags can become unwieldy. Asynchronous frameworks like Embassy provide a modern, high-level alternative.

- **Core Pattern:** Use an async runtime like Embassy to manage concurrency. Embassy leverages Rust's async/.await syntax to enable cooperative multitasking on a single thread without a traditional RTOS. async functions are defined as tasks, and an executor polls their Futures, running them to completion.[62]

This pattern represents a significant evolution in embedded concurrency. Traditional approaches often involve either manually coding complex, event-driven state machines or using a preemptive RTOS, which introduces overhead like separate stacks for each task and context switching costs.[65] Embassy uses a compile-time approach, transforming

async functions into efficient state machines that are managed by a lightweight, cooperative executor. A key feature of this pattern is its power efficiency: when no tasks are ready to run (i.e., all tasks are .awaiting an event like a timer or a GPIO interrupt), the executor can automatically put the processor into a low-power sleep state. The processor is then woken

directly by the hardware interrupt that makes a task ready to run again. This combines the ergonomic benefits of writing sequential-looking, threaded code with the performance and power efficiency of a hand-tuned, event-driven system.[62]

Part III: Systems Programming at the Bare Metal

This final part addresses the lowest level of the Rust programming landscape: the development of operating systems and kernel modules. This domain requires direct interaction with hardware, meticulous memory management, and the careful handling of unsafe code. Here, Rust's safety guarantees are not a convenience but a powerful tool for building reliable and secure low-level software.

The defining characteristic of idiomatic low-level Rust is not the pervasive use of unsafe but the disciplined construction of a **safe/unsafe boundary**. While operations like dereferencing raw pointers, writing to memory-mapped I/O registers, or calling foreign functions in C are inherently unsafe—meaning the compiler cannot verify their memory safety—the idiomatic approach is to encapsulate these operations within safe abstractions.[59] A function or method will use an

unsafe block internally to perform the necessary low-level operation, but its public API will be safe. The implementation of this safe API carries the burden of upholding the invariants required to make the internal unsafe block valid. This contract with the caller is documented explicitly with a // SAFETY: comment preceding the unsafe block, explaining why the operation is sound in that context.[67] This architectural pattern forces developers to minimize the surface area of

unsafe code and to clearly reason about and document its correctness. The result is that the majority of the kernel or driver logic can still be written in safe Rust, benefiting from the full power of the compiler's static analysis.

Concurrency Pattern	Environment	Granularity	Primary Use Case	Key Trade-off
std::thread + Arc<Mutex<T>>	std	Thread	Shared-state concurrency	Performance (lock contention) vs. direct access

mpsc Channels	std	Task	Message passing	Simplicity/decoupling vs. data copy overhead
Critical Sections	no_std	Data	Interrupt safety (complex data)	Simplicity vs. increased interrupt latency
Atomics	core / no_std	Data	Interrupt safety (simple data)	High performance vs. limited to primitive ops
Async (Tokio)	std	Task	I/O-bound concurrency	High scalability vs. runtime complexity
Async (Embassy)	no_std	Task	Embedded concurrency	Power efficiency/ergonomics vs. cooperative model

Pattern Suite 3.1: Idioms for Rust in the Linux Kernel

The integration of Rust as a second language into the Linux kernel is an ongoing effort that has established a set of specific idioms and guidelines for writing kernel-level code.

Safe Abstractions over unsafe C FFI

Interaction with the vast existing C codebase of the kernel is a primary requirement. This is managed through a Foreign Function Interface (FFI).

- **Core Pattern:** When calling C functions or accessing C data structures, use the specific

type aliases provided by the kernel prelude (e.g., c_int), not the generic ones from core::ffi, to ensure correct type mapping.[67] Raw pointers and FFI function calls, which are inherently
unsafe, should be immediately wrapped within higher-level, safe Rust abstractions. For instance, a kernel object managed by C functions for allocation and deallocation should be wrapped in a Rust struct that implements the Drop trait to ensure the deallocation function is always called, preventing resource leaks.

This pattern is a direct application of the safe/unsafe boundary principle. The goal is to contain the unsafe FFI interactions within a minimal layer of code, presenting a safe, idiomatic Rust API to the rest of the Rust kernel module.

Wrapping Kernel Primitives: Naming and Design

To ensure that the Rust code is understandable to long-time kernel developers, a consistent naming convention is employed when creating abstractions over existing C components.

- **Core Pattern:** When wrapping a C macro, function, or constant, the Rust abstraction should use a name that is as close as reasonably possible to the original C name. However, the casing should be adapted to follow Rust's idiomatic conventions (e.g., snake_case for functions, PascalCase for types). For example, C constants like #define GPIO_LINE_DIRECTION_IN 0 would be wrapped in a Rust enum as pub enum LineDirection { In =... } within a gpio module.[67]

This pattern creates a clear and predictable mapping between the C and Rust worlds, reducing the cognitive friction for developers who must work with both.

Pattern Suite 3.2: Architectural Patterns for OS Development

The development of entire operating systems in Rust, pioneered by projects like Philipp Oppermann's blog_os and Redox OS, has given rise to several high-level architectural patterns.

Bootstrapping a Freestanding Binary

The very first step in writing an OS is to create an executable that can run on bare metal without any underlying support.

- **Core Pattern:** A kernel is built as a freestanding binary by using the #![no_std] and #![no_main] crate attributes. This removes the dependency on the standard library and the standard Rust runtime. The developer must then provide a custom entry point (often a function named _start specified to the linker) and a panic handler function (marked with the #[panic_handler] attribute) that defines the system's behavior in the event of an unrecoverable error.[42] This pattern provides the developer with complete control over the machine's state from the very first instruction executed by the CPU.

Memory Management: Paging and Heap Implementation

Once the kernel is running, it must establish control over the system's memory.

- **Core Pattern:** The first step in memory management is typically to set up paging, which creates a virtual address space that isolates the kernel from other potential processes and provides a clean memory layout. This involves creating page tables that map virtual addresses to physical memory frames. Once a region of virtual memory has been designated for the kernel's heap, a heap allocator must be initialized. This follows the pattern described in Suite 2.2: an object implementing the GlobalAlloc trait is created and registered with #[global_allocator], making dynamic allocation via Box and Vec possible within the kernel.[51]

Rust's safety features provide significant advantages in this notoriously error-prone domain. Wrapper types can be created for page tables, page table entries, and physical memory frames. The ownership and borrow checking rules can then be used to prevent common errors, such as using a physical frame after it has been freed or creating conflicting mappings for the same virtual page.

Redox OS: "Everything is a URL"

Redox OS, a microkernel-based operating system written entirely in Rust, showcases a unique and powerful architectural pattern for resource management.

- **Core Pattern:** In Redox OS, every resource—including files, network sockets, device drivers, and inter-process communication channels—is represented as a URL-like "scheme." To open a file, a process opens file:/path/to/file; to make a TCP connection, it opens tcp:127.0.0.1:80.[69] Device drivers are implemented as userspace processes that

register themselves with the kernel as the handler for a specific scheme (e.g., a disk driver might handle the
disk: scheme).

This "everything is a URL" pattern, inspired by the Plan 9 operating system, creates a uniform interface for all resource interactions. When combined with Redox's microkernel design, it yields significant benefits for security and stability. Since drivers are just regular userspace processes, a bug or crash in a device driver cannot crash the kernel or corrupt kernel memory; it only terminates the driver process. This is a fundamental architectural departure from monolithic kernels like Linux, where a faulty driver runs in kernel space and can bring down the entire system.[69]

Conclusion

This layered exploration of idiomatic Rust reveals a language whose patterns are a direct and consistent expression of its core design philosophy. Across every layer, from high-level applications to the bare metal of the kernel, the principles of safety, explicit control, and zero-cost abstraction are not merely present but are the primary forces shaping how developers write effective and efficient code.

In the std environment, the idioms focus on leveraging the safe, high-level abstractions provided by the standard library. Patterns like the Result/Option dichotomy, the ? operator, and the Arc<Mutex<T>> construct demonstrate how Rust transforms potential runtime errors and data races into compile-time verifiable properties of the type system. High-level features like iterators and async/.await provide expressive and ergonomic APIs without sacrificing the low-level performance expected of a systems language.

As environmental constraints increase in the no_std world, the patterns shift towards explicit resource management and portability. The layered design of core, alloc, and std allows developers to make conscious trade-offs, opting into capabilities like heap allocation only when required and available. Patterns like the heapless crate for stack-based collections and the embedded-hal trait-based driver model showcase how Rust's type system can be used to enforce determinism and create a portable, reusable ecosystem even in the absence of a full standard library.

At the deepest layer of kernel and OS development, the central pattern becomes the disciplined management of the safe/unsafe boundary. Idiomatic Rust does not eschew unsafe code but contains it within safe abstractions, minimizing its scope and demanding rigorous justification for its use. Architectural patterns seen in projects like blog_os and Redox OS demonstrate that Rust's safety guarantees are not a hindrance but a powerful asset for

building reliable and secure low-level systems from the ground up.

Ultimately, the journey through Rust's layers reveals a remarkable consistency. The same fundamental tools—ownership, borrowing, traits, and the type system—are applied with increasing precision to solve the problems of each domain. An idiomatic Rust developer is one who understands not just the patterns themselves, but the underlying principles that give rise to them, enabling them to write code that is not only correct and performant but also a clear and robust expression of its design intent.

Works cited

1. Applying Design Patterns in Rust | Learn About Rust Programming - Electro4u, accessed on September 6, 2025, https://electro4u.net/blog/applying-design-patterns-in-rust--learn-about-rust-programming--4243
2. Fearless Concurrency - The Rust Programming Language, accessed on September 6, 2025, https://doc.rust-lang.org/book/ch16-00-concurrency.html
3. Crate std - Rust Documentation, accessed on September 6, 2025, https://doc.rust-lang.org/std/
4. std - Rust, accessed on September 6, 2025, https://www.cs.brandeis.edu/~cs146a/rust/doc-02-21-2015/std/index.html
5. std - Rust, accessed on September 6, 2025, https://stdrs.dev/
6. Standard Library - Comprehensive Rust - Google, accessed on September 6, 2025, https://google.github.io/comprehensive-rust/std-types/std.html
7. Std library types - Rust By Example, accessed on September 6, 2025, https://doc.rust-lang.org/rust-by-example/std.html
8. Error Handling - Idiomatic Rust Snippets, accessed on September 6, 2025, https://idiomatic-rust-snippets.org/essentials/core-concepts/error-handling.html
9. Error Handling - The Rust Programming Language - MIT, accessed on September 6, 2025, https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/error-handling.html
10. Mastering Error Handling in Rust: Result, Option, and Match Explained - Djamware, accessed on September 6, 2025, https://www.djamware.com/post/6881abbd2d906261a96f3386/mastering-error-handling-in-rust-result-option-and-match-explained
11. Recoverable Errors with Result - The Rust Programming Language - Rust Documentation, accessed on September 6, 2025, https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html
12. Item 4: Prefer idiomatic Error types - Effective Rust, accessed on September 6, 2025, https://effective-rust.com/errors.html
13. My First Open Source Project (I) — Rust Error Handling and axum-error-handler - Medium, accessed on September 6, 2025, https://medium.com/@david99900/my-first-open-source-project-i-rust-error-handling-and-axum-error-handler-4905d551388a

14. std::collections - Rust, accessed on September 6, 2025, https://doc.rust-lang.org/std/collections/index.html
15. HashMap in std::collections - Rust Documentation, accessed on September 6, 2025, https://doc.rust-lang.org/std/collections/struct.HashMap.html
16. Using Vec instead of HashMap? : r/rust - Reddit, accessed on September 6, 2025, https://www.reddit.com/r/rust/comments/10k3u52/using_vec_instead_of_hashmap/
17. Mastering Rust Arc and Mutex: A Comprehensive Guide to Safe Shared State in Concurrent Programming | by Syed Murtza | Medium, accessed on September 6, 2025, https://medium.com/@Murtza/mastering-rust-arc-and-mutex-a-comprehensive-guide-to-safe-shared-state-in-concurrent-programming-1913cd17e08d
18. Mutex in std::sync - Rust, accessed on September 6, 2025, https://doc.rust-lang.org/std/sync/struct.Mutex.html
19. Arc in std::sync - Rust Documentation, accessed on September 6, 2025, https://doc.rust-lang.org/std/sync/struct.Arc.html
20. Shared-State Concurrency - The Rust Programming Language, accessed on September 6, 2025, https://doc.rust-lang.org/book/ch16-03-shared-state.html
21. Mutex, Send and Arc - 100 Exercises To Learn Rust, accessed on September 6, 2025, https://rust-exercises.com/100-exercises/07_threads/11_locks.html
22. Is there an alternative to Arc
23. Introduction to I/O in Rust. What is I/O? | by Murad Bayoun - Medium, accessed on September 6, 2025, https://medium.com/@bayounm95.eng/introduction-to-i-o-in-rust-88a247e0f156
24. std::io - Rust, accessed on September 6, 2025, https://doc.rust-lang.org/std/io/index.html
25. Comparing Performance: Loops vs. Iterators - The Rust Programming Language, accessed on September 6, 2025, https://doc.rust-lang.org/book/ch13-04-performance.html
26. Processing a Series of Items with Iterators - The Rust Programming ..., accessed on September 6, 2025, https://doc.rust-lang.org/book/ch13-02-iterators.html
27. Compositional Reasoning about Advanced Iterator Patterns in Rust - Research Collection, accessed on September 6, 2025, https://www.research-collection.ethz.ch/bitstreams/30a5624b-b73f-4f5e-8877-88d65b76e7cc/download
28. Rust Tutorial – Learn Advanced Iterators & Pattern Matching by Building a JSON Parser, accessed on September 6, 2025, https://www.freecodecamp.org/news/rust-tutorial-build-a-json-parser/
29. Tutorial | Tokio - An asynchronous Rust runtime, accessed on September 6, 2025, https://tokio.rs/tokio/tutorial
30. How to Use Tokio with Rust. Practical guide to asynchronous... - Altimetrik Poland Tech Blog, accessed on September 6, 2025, https://altimetrikpoland.medium.com/how-to-use-tokio-with-rust-f42a56cbd720
31. Shared state | Tokio - An asynchronous Rust runtime, accessed on September 6, 2025, https://tokio.rs/tokio/tutorial/shared-state

32. Tokio | 03 , Shared State | Mutex | Rust | by Mike Code - Medium, accessed on September 6, 2025, https://medium.com/@mikecode/tokio-03-shared-state-mutex-rust-e546801472ab
33. axum::error_handling - Rust - Docs.rs, accessed on September 6, 2025, https://docs.rs/axum/latest/axum/error_handling/index.html
34. axum::error_handling - Rust - Straw Lab, accessed on September 6, 2025, https://strawlab.org/strand-braid-api-docs/latest/axum/error_handling/index.html
35. Using Rust with Axum for error handling - LogRocket Blog, accessed on September 6, 2025, https://blog.logrocket.com/rust-axum-error-handling/
36. Stateful Web with Axum - Rust Development Classes, accessed on September 6, 2025, https://rust-classes.com/chapter_7_2
37. axum - Rust - Docs.rs, accessed on September 6, 2025, https://docs.rs/axum/latest/axum/
38. State in axum::extract - Rust - Docs.rs, accessed on September 6, 2025, https://docs.rs/axum/latest/axum/extract/struct.State.html
39. [Noob] What exactly is #![no_std], and why is it so useful sometimes to be without it? : r/rust - Reddit, accessed on September 6, 2025, https://www.reddit.com/r/rust/comments/9eyc21/noob_what_exactly_is_no_std_and_why_is_it_so/
40. Item 33: Consider making library code no_std compatible - Effective ..., accessed on September 6, 2025, https://effective-rust.com/no-std.html
41. Can Rust code compile without the standard library? - Stack Overflow, accessed on September 6, 2025, https://stackoverflow.com/questions/61034534/can-rust-code-compile-without-the-standard-library
42. The smallest #![no_std] program - The Embedonomicon, accessed on September 6, 2025, https://docs.rust-embedded.org/embedonomicon/smallest-no-std.html
43. no_std - The Embedded Rust Book, accessed on September 6, 2025, https://docs.rust-embedded.org/book/intro/no-std.html
44. When adding `#![no_std]` to a library, are there any disadvantages or complications for the users of that library? - Stack Overflow, accessed on September 6, 2025, https://stackoverflow.com/questions/57611219/when-adding-no-std-to-a-library-are-there-any-disadvantages-or-complicati
45. PSA for `std` Feature in `no_std` Libraries : r/rust - Reddit, accessed on September 6, 2025, https://www.reddit.com/r/rust/comments/1hs6spy/psa_for_std_feature_in_no_std_libraries/
46. No_std with feature flag : two ways? - help - The Rust Programming Language Forum, accessed on September 6, 2025, https://users.rust-lang.org/t/no-std-with-feature-flag-two-ways/90139
47. Practical guides on no_std and wasm support - tutorials - Rust Users Forum, accessed on September 6, 2025, https://users.rust-lang.org/t/practical-guides-on-no-std-and-wasm-support/947

62
48. Advanced Rust Patterns for Embedded Programming - Enhance Your Skills - MoldStud, accessed on September 6, 2025, https://moldstud.com/articles/p-advanced-rust-patterns-for-embedded-programming-enhance-your-skills
49. Collections - The Embedded Rust Book, accessed on September 6, 2025, https://docs.rust-embedded.org/book/collections/
50. Nine Rules for Running Rust on Embedded Systems - Medium, accessed on September 6, 2025, https://medium.com/data-science/nine-rules-for-running-rust-on-embedded-systems-b0c247ee877e
51. Heap Allocation | Writing an OS in Rust, accessed on September 6, 2025, https://os.phil-opp.com/heap-allocation/
52. Rust Custom Allocators - Ian Bull, accessed on September 6, 2025, https://ianbull.com/posts/rust-custom-allocators/
53. Turns out, using custom allocators makes using Rust way easier - Reddit, accessed on September 6, 2025, https://www.reddit.com/r/rust/comments/1jlopns/turns_out_using_custom_allocators_makes_using/
54. embedded-hal - Comprehensive Rust - Google, accessed on September 6, 2025, https://google.github.io/comprehensive-rust/bare-metal/microcontrollers/embedded-hal.html
55. A Hardware Abstraction Layer (HAL) for embedded systems - GitHub, accessed on September 6, 2025, https://github.com/rust-embedded/embedded-hal
56. Portability - The Embedded Rust Book, accessed on September 6, 2025, https://docs.rust-embedded.org/book/portability/
57. Writing embedded drivers in Rust isn't that hard. Part 1 - Henrik Böving, accessed on September 6, 2025, https://hboeving.dev/blog/rust-2c-driver-p1/
58. rust-embedded/awesome-embedded-rust: Curated list of resources for Embedded and Low-level development in the Rust programming language - GitHub, accessed on September 6, 2025, https://github.com/rust-embedded/awesome-embedded-rust
59. Is anyone using Rust for embedded work - Reddit, accessed on September 6, 2025, https://www.reddit.com/r/embedded/comments/1amh6jq/is_anyone_using_rust_for_embedded_work/
60. Demystifying Rust Embedded HAL Split and Constrain Methods, accessed on September 6, 2025, https://blog.theembeddedrustacean.com/demystifying-rust-embedded-hal-split-and-constrain-methods
61. Concurrency - The Embedded Rust Book, accessed on September 6, 2025, https://docs.rust-embedded.org/book/concurrency/
62. embassy-rs/embassy: Modern embedded framework, using ... - GitHub, accessed on September 6, 2025, https://github.com/embassy-rs/embassy
63. Embassy, accessed on September 6, 2025, https://embassy.dev/

64. How Rust & Embassy Shine on Embedded Devices (Part 1) | Carl M. Kadie & Brad Gibson, accessed on September 6, 2025, https://medium.com/@carlmkadie/how-rust-embassy-shine-on-embedded-devices-part-1-9f4911c92007
65. Concurrency in Embedded Systems: Rust and Async Solutions - Witekio, accessed on September 6, 2025, https://witekio.com/blog/rust-async-embedded-systems/
66. The Pragmatic Future of the Linux Kernel: Balancing Rust and C | by Noah Bean - Medium, accessed on September 6, 2025, https://medium.com/@noahbean3396/the-pragmatic-future-of-the-linux-kernel-balancing-rust-and-c-e518117328b7
67. Coding Guidelines — The Linux Kernel documentation, accessed on September 6, 2025, https://docs.kernel.org/rust/coding-guidelines.html
68. Writing an OS in Rust, accessed on September 6, 2025, https://os.phil-opp.com/
69. Programming for Redox OS - DEV Community, accessed on September 6, 2025, https://dev.to/legolord208/programming-for-redox-os-4124
70. Redox OS, accessed on September 6, 2025, https://www.redox-os.org/
71. redox-os/redox: Mirror of https://gitlab.redox-os.org/redox-os/redox - GitHub, accessed on September 6, 2025, https://github.com/redox-os/redox