Project guidelines for panics, error handling, and build-time failures

Alexandre Courbot

Kangrejos 2026

Goals

- Discuss project-wide guidelines for:
  - Which error handling patterns are considered idiomatic in R4L.
  - When panicking (explicitly or implicitly) is acceptable.
- Identify the already-accepted practice and what is still undecided.

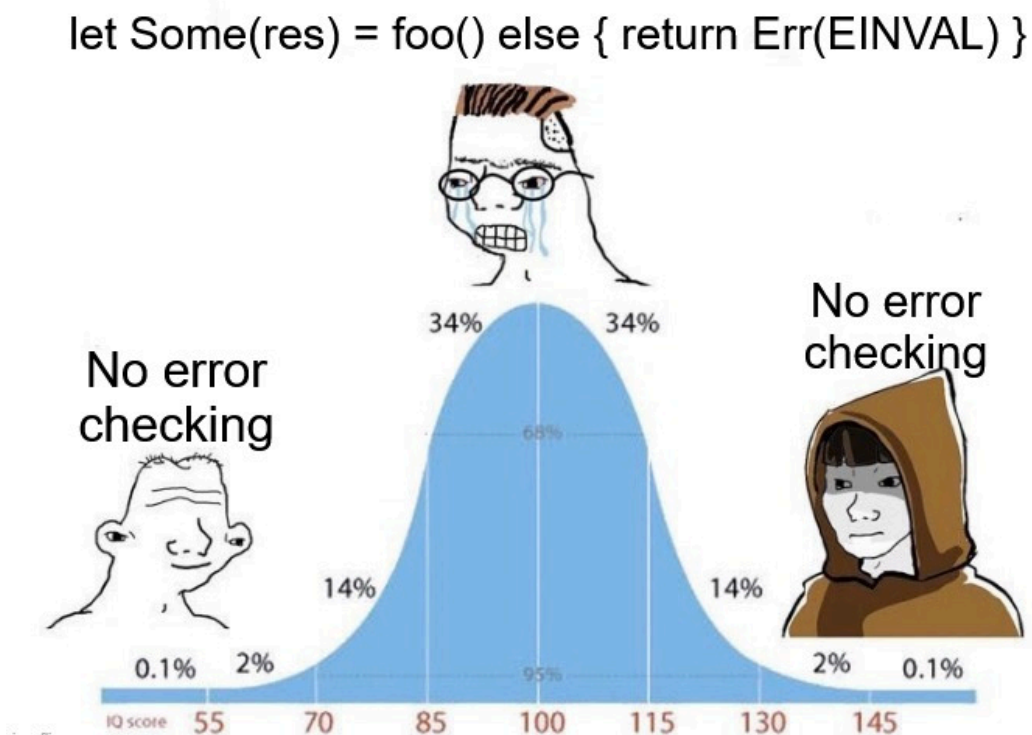
Desired outcome: a few more lines in `coding-guidelines.rst`.

The ladder of error checking

- Nowhere, failure is impossible 😎
- Compiler, guaranteed
  - `static_assert!`, `const_assert!`
- Compiler, best-effort
  - `build_assert!`
- Runtime
  - `Error` and `Result`
  - Panic 😱

Remove failure as an option

Make errors impossible in the first place.



Remove failure as an option

Use enums, `NonZero`, `Bounded`, etc. to make the type system carry desired invariants.

```
struct VgpuState {
    enabled: bool,
    total_vfs: u16,
}


// size_of::<VgpuState>() == 4
```

```
enum VgpuState {
    Disabled,
    Enabled {
        total_vfs: NonZero<u16>,
    },
}
// size_of::<VgpuState>() == 2
```

Build-time errors

Guaranteed: `static_assert!`, `const_assert!`

Statically check input at construction:

```
let v: NonZero<u16> = cv!(1);
```

Best-effort: `build_assert!`

- Relies on optimizer; more fragile.
- Requires `#[inline(always)]` if used on a function argument.

```
Bounded::<u64, 2>::from_expr(val & 0x3)
```

Runtime errors

`Error` and `Result` (or `Option`):

- Adds complexity: caller needs to check them
  - More convoluted code
  - Cognitive load for reader
- Inevitable with external input

Trend:

- Rust provides opportunities to move these to build-time
- The temptation to get rid of them introduces dilemmas

Avoiding runtime errors: unreachable code

```
// `irq_type` is never `Intx`.
fn rearm_method(irq_type: IrqType) ->
Option<Rearm> {
    match irq_type {
        IrqType::Intx => None,
        IrqType::Msi =>
            Some(Rearm::Serviced),
        IrqType::MsiX =>
            Some(Rearm::Subtree),
    }
}
```

```
enum MsiType { Msi, MsiX }

fn rearm_method(irq_type: MsiType) ->
Rearm {
    match irq_type {
        MsiType::Msi => Rearm::Serviced,
        MsiType::MsiX => Rearm::Subtree,
    }
}
```

Impossible should mean no failure path: validate data once at build-time or runtime, then work with infallible state.

Panics! 😱

Really the one thing we want to avoid.

When they are used:

- Bug (i.e. invalid state) detection
- “This won’t really happen” (famous last words)

```
Documentation/rust/coding-guidelines.rst:

Please note that panicking should be very rare and used only with a good
reason. In almost all cases, a fallible approach should be used, typically
returning a `Result`.
```

Panics! 😱

The problem with panicking:

- Bad optics (“I thought Rust was safe!”)
- If triggered from user-space → CVE
- Each site is a potential landmine
  - Many get optimized away, but …
  - … still a wart in the source code and cognitive load on readers
  - … code evolves. What is obviously correct today might not be tomorrow

Panics! 😱

Context matters:

- Core code is more controlled...
- ... but do we trust device drivers as much?

A few real-life examples

For which I couldn’t find a rule that guided the decision.

Note how all of these introduce a panic to avoid returning a `Result`.

Erroneous Behavior (EB)

A fallback for a caller bug. Panics if debug assertions are enabled.

```
/// ... it is considered a bug to call this function with an out-of-range
/// value.
pub fn udelay(delta: Delta) {
    debug_assert!(delta.as_nanos() >= 0);
    debug_assert!(delta <= MAX_UDELAY_DELTA);

    // Clamp and proceed.
```

Alternatives:

- Return a `Result`
- Restrictive input type enforcing the precondition
- Just loop?

Invariant violations

A caller bug with no recoverable path.

```
    pub(crate) fn add_death(&self, death: ...) {
        assert!(
            core::ptr::eq(self, &**death.node),
            "attempt to add NodeDeath to the wrong death list"
        );
```

Alternatives:

- Return a `Result`
- Make `unsafe` and document UB
- Erroneous Behavior? (guaranteed to make things worse down the road)
- Poison the driver without panicking (introduces new driver state)

Conditional panics

```
impl Bitmap {
    pub fn set_bit(&mut self, index: usize) {
        bitmap_assert_return!(index < self.len(), ...);
    ...
    }
}
```

`bitmap_assert_return!`:

- `assert!` if `CONFIG_RUST_BITMAP_HARDENED` is set
- `pr_err!` and returns otherwise

Alternatives:

- Return a `Result`
- Embrace the EB and remove the config knob

Context-conditional panics

Build-time failure in const context, panic in non-const context.

```
impl Hertz {
    const GHZ_TO_HZ: c_ulong = 1_000_000_000;
    // pub type c_ulong = usize;
    pub const fn from_ghz(ghz: c_ulong) -> Self {
        Self(ghz * Self::GHZ_TO_HZ)
    }
}
```

Alternatives:

- Return a `Result`
- Introduce documented EB
- `#[const_eval_only]` to enforce const-context only code

The dilemma

The code can reach an invalid state that you cannot make unrepresentable.

Do you:

- Return a `Result`?
- Panic?
- Continue with Erroneous Behavior and maybe a warning?

Implicit panics

Little landmines silently sprinkled by the compiler.

Many can be avoided:

```
if i >= v.len() {
    return Err(EINVAL)
}
v[i]
```

```
v.get(i).ok_or(EINVAL)
```

Runtime cost if not optimized away: instructions for test + jump.

Problem: they are not easily detected in the code.

Implicit panics in `nova-core`

(`CC_OPTIMIZE_FOR_PERFORMANCE=y`, `RUST_OVERFLOW_CHECKS=y`)

```
Panic sites per kind:
    65  panic_const_add_overflow
    61  panic_const_sub_overflow
    19  panic_const_mul_overflow
     8  panic_bounds_check
     7  unwrap_failed
     2  panic
     1  expect_failed
     1  assert_failed::<&[u8], &[u8]>
     1  assert_failed::<usize, usize>
     1  assert_failed_inner
     1  panic_const_shl_overflow
     1  panic_fmt
nova_core.o: 170 panic sites in 52 of 294 functions
```

Implicit panics from `core`

`kernel::module_param::set_param`: one panic site coming from an inlined `CStr::from_ptr`:

```
impl CStr {
    pub const unsafe fn from_ptr<'a>(ptr: *const c_char) -> &'a CStr {
        // SAFETY: The caller has provided a pointer that points to a valid C
        // string with a NUL terminator less than `isize::MAX` from `ptr`.
        let len: usize = unsafe { strlen(ptr) };

        // ...
        slice::from_raw_parts(ptr.cast(), len + 1)
    }
}
```

Conclusion

The agreed-on (unwritten consensus):

- No error > compile-time errors > runtime errors
  - But external input always gets runtime-checked
  - Never make infallible code fallible
- A reachable panic is a bug

Conclusion

The unknown:

- When is Erroneous Behavior tolerated to avoid a `Result`?
  - Logged?
  - Panic if debug assertions are enabled?
- When is panicking tolerated to avoid a `Result`?
  - Invariant violations?
  - Implicit panics in verified code paths?
- Should we monitor and eliminate implicit panics?
- Same rules for core and drivers? Kernel-wide or per-subsystem?