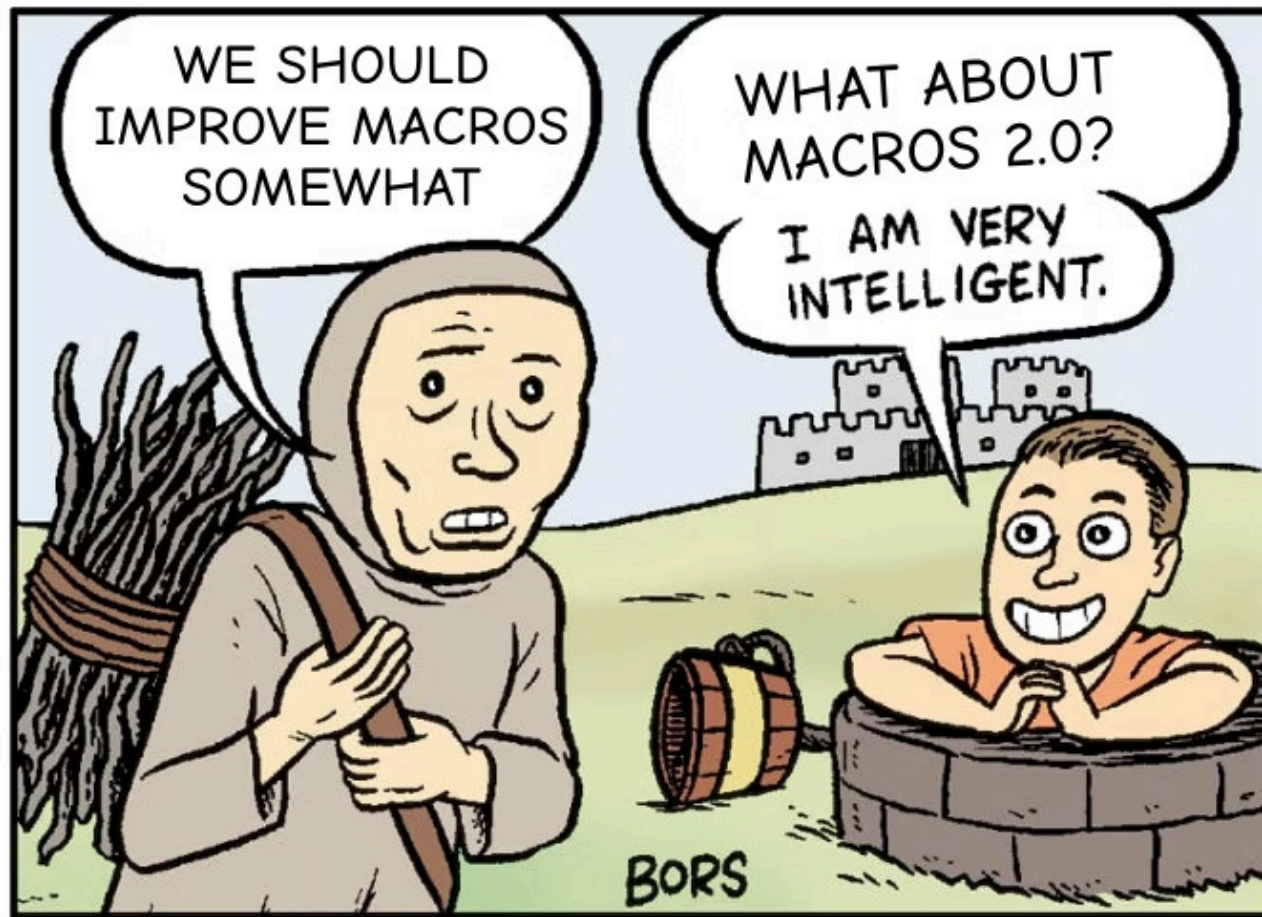


# Rust Declarative Macro Improvements: A Step forward for the Rust macros usability

Today we'll explore the most significant improvement to Rust's macro system since 1.0 - the extension of declarative macros to support attributes and derives. This isn't just about making macro writing easier; it's about removing one of the biggest obstacles to Rust adoption in systems programming, particularly for Rust for Linux development.

Author: Vincenzo Palazzo [vincenzopalazzodev@gmail.com](mailto:vincenzopalazzodev@gmail.com)

Co-Leading wg-macros on the rust compiler



9-13-16

DIST. BY UNIVERSAL UCLICK

# The Current Reality

If you want to create a custom derive macro or attribute macro in Rust today, you **must** write a procedural macro. There's no way around it.

## Separate Crate Required

Every proc macro needs its own compilation unit with special configuration

## Heavy Dependencies

syn (25,000 lines), quote (5,000 lines),  
proc-macro2 (15,000 lines)

## Build Time Impact

Proc macros make incremental rebuilds slower

# Kernel Development Pain Points

In kernel space, procedural macros create unique challenges that break traditional development workflows.

01

---

## Dependency Explosion

45,000 lines of parsing code just to use a derive macro. Every line increases attack surface and requires security review.

02

---

## Build Time Regression

Kernel developers depending on how you build them.

03

---

## Maintenance Overhead

Separate macro crates complicate build systems, review processes, and ongoing maintenance for simple modules.

## A Concrete Example: module! Macro

In Rust for Linux, every kernel module requires this macro. Currently implemented as a procedural macro, it forces every compilation to build syn, quote, and proc-macro2 first.

```
module! {  
    type: MyModule,  
    name: "my_kernel_module",  
    license: "GPL",  
}
```

This simple macro generates module initialisation code but requires attribute-like behaviour. The result? Massive compilation overhead for basic functionality that has nothing to do with the kernel itself.

# But `macro_rules!` Are Actually Brilliant

## What They Do Well

- Compile quickly with minimal overhead
- Excellent hygiene with `$crate` mechanism - it prevents name conflicts and ensures macros work correctly across crate boundaries.
- Can define and use a macro in the same crate. No separate crate required.
- Battle-tested reliability and performance

## The Only Problem

They've been limited to function-like macros (`mac!(...)`). No `derives`, no attributes, no way to replace procedural macros for the most common use cases.

✔ Until now.

# Three RFCs Change Everything

1

## RFC 3697

Attribute macro support with `attr()` rule syntax

2

## RFC 3698

Derive macro support with `derive()` rule syntax

3

## RFC 3715

Safety marking for unsafe macros requiring explicit acknowledgement

These aren't theoretical proposals. The first two are already implemented in nightly Rust and available for testing today.

# RFC 3697: Declarative Attribute Macros

Implementation <https://github.com/rust-lang/rfcs/pull/3697> by [Josh Triplett](#)

Approved and merged in August 2025, this RFC introduces the ability to write attribute macros using `macro_rules!` through the new `attr()` rule syntax.

```
macro_rules! main {  
  attr() ($func:item) => {  
    make_async_main!($func)  
  };  
  attr(threads = $threads:literal) ($func:item) => {  
    make_async_main!($threads, $func)  
  };  
}
```

Notice the structure: `attr()` matches attribute arguments, followed by parentheses matching what the attribute applies to. Completely backwards compatible with existing function-like macro calls.



# Attribute Syntax in Action

## Simple Usage

```
#[main]
async fn main() {
    // Your async code here
}
```

## With Arguments

```
#[main(threads = 4)]
async fn main() {
    // Runs with 4 threads
}
```

The brilliance lies in the backwards compatibility. Add attr rules to existing macros without breaking anything. Function-like calls still work; attribute usage is now possible too.

# Complex Attribute Example: Benchmarking

Here's a practical example that would have required a full procedural macro before:

```
macro_rules! bench {
  attr() ($func:item) => {
    #[cfg(test)]
    mod bench {
      use super::*;

      $func

      #[test]
      fn benchmark() {
        let start = std::time::Instant::now();
        // Call the function multiple times
        println!("Benchmark took: {:?}", start.elapsed());
      }
    }
  };
}
```

```
#[bench]
fn foo() {
  /* Compute-intensive code here */
}
```

Now it's just a few lines of declarative macro code. No syn, no quote, no separate crate. Compiles instantly with full hygiene support.



# RFC 3698: Declarative Derive Macros

Implementation <https://github.com/rust-lang/rfcs/pull/3698> by [Josh Triplett](#)

Equally revolutionary, this RFC introduces `derive()` rule syntax for implementing derive macros with `macro_rules!`.

```
#![feature(macro_derive)]
macro_rules! Answer {
    derive() (struct $name:ident $_:tt) => {
        impl Answer for $name {
            fn answer(&self) -> u32 { 43 }
        }
    };

    derive() (enum $name:ident $_:tt) => {
        impl Answer for $name {
            fn answer(&self) -> u32 { /* ... */ }
        }
    };
}
```

# Derive Usage and Conventions

## Standard Syntax

```
#[derive(Answer)]  
struct MyStruct;
```

```
#[derive(Answer)]  
enum MyEnum {  
    A, B, C  
}
```

## Key Insight

The macro has the same name as the trait. When you write `#[derive(Answer)]`, it looks for a macro named `Answer` with `derive()` rules.

Follows established Rust conventions whilst maintaining tool compatibility.

Playground code: <https://play.rust-lang.org/?version=nightly&mode=debug&edition=2024&gist=04910934860c1b535513c2d2c50a1433>

# RFC 3715: Unsafe Derives and Attributes

Implementation <https://github.com/rust-lang/rfcs/pull/3715h> by [Josh Triplett](#)

Some derives and attributes can cause undefined behaviour if used incorrectly. This RFC introduces explicit safety marking for potentially dangerous macros.

## Unsafe Derive

```
macro_rules! FromBytes {
    unsafe derive() ($item:item) => {
        // Implementation assumes no padding
    };
}

#[derive(unsafe(FromBytes))]
#[repr(C)]
struct PackedData {
    a: u32,
    b: u32,
}
```

## Unsafe Attribute


```
macro_rules! no_mangle_export {
    unsafe attr() ($func:item) => {
        #[no_mangle]
        pub unsafe extern "C" $func
    };
}

#[unsafe(no_mangle_export)]
fn my_function() { /* ... */ }
```

## Final Reasons

Some traits place requirements on implementations that the Rust compiler cannot verify. Those traits can mark themselves as `unsafe`, requiring `unsafe impl` syntax to implement. However, trait `derive` macros cannot currently require `unsafe`.

This brings macro safety in line with Rust's other safety mechanisms. The RFC is approved but not yet implemented; when it lands, it will complete the safety story for declarative macros.

 RFC 3715 status: Not yet approved. Need feedback from potential users on use cases

# Current Limitations and Roadmap

1

## Parsing Flexibility

Declarative macros can't parse Rust syntax as flexibly as procedural macros. Complex generic bounds may struggle where syn succeeds.

2

## The 80/20 Rule

Most macros don't need that flexibility. 80% of derive and attribute macros do simple transformations that declarative macros handle perfectly.

3

## Fragment Fields (RFC 3714)

Under evaluation feature will allow accessing parsed components like `$field.name` and `$field.type`, closing remaining gaps.



# Future Fragment Fields Syntax

Proposal <https://github.com/rust-lang/rfcs/pull/3714> by [Josh Triplett](#)

RFC 3714 will enable much more sophisticated field manipulation:

```
macro_rules! get_name {  
    ($t:adt) => { stringify!($ {t.name}) }  
}  
  
fn main() {  
    let n1 = get_name!(struct S { field: u32 });  
    let n2 = get_name!(enum E { V1, V2 = 42, V3(u8) });  
    let n3 = get_name!(union U { u: u32, f: f32 });  
    println!("{n3}{n1}{n2}"); // prints "USE"  
}
```

2024

1

Answering to the question if it is easy to implement these things in rust internal

2

2025 Q2

Make some experiment around the compiler

2025 Q4

3

See how people will use these features and pay attention to the concern that Josh highlights on the

<https://github.com/rust-lang/rfcs/blob/master/text/3697-declarative-attribute-macros.md#drawbacks>

# Kernel Development Constraints

Understanding why these improvements are crucial requires grasping the unique constraints of kernel development:



# Declarative Solution: Kernel module! Macro

Here's how we could reimplement the kernel's module! macro using declarative attributes:

```
macro_rules! module {
  attr(
    name = $name:literal,
    author = $author:literal,
    license = $license:literal
  )(
    struct $type:ident;
  ) => {
    struct $type;

    #[no_mangle]
    pub extern "C" fn init_module() -> i32 {
      printk!("Loading module: {}\n", $name);
      <$type as $crate::Module>::init()
    }

    #[no_mangle]
    pub extern "C" fn cleanup_module() {
      printk!("Unloading module: {}\n", $name);
      <$type as $crate::Module>::exit()
    }

    const MODULE_LICENSE: &[u8] = $license.as_bytes();
    const MODULE_AUTHOR: &[u8] = $author.as_bytes();
  };
}

#[module(name = "DeclarativeModule", author = "Vincenzo Palazzo", license = "MIT")]
struct NewModule;

fn main() {}
```

## Dramatic Performance Improvements

0

### **syn Dependency**

Eliminates 25,000 lines of  
parsing code

0

### **quote Dependency**

Removes 5,000 lines of code  
generation

0

### **proc-macro2**

Saves 15,000 lines of token  
manipulation

??%

### **Build Time Reduction**

Clean build times for modules  
using proc macros

More importantly, incremental builds return to sub-second performance. Change your module code and you're back to the rapid iteration kernel developers expect.

# Community Impact

The benefits extend beyond technical improvements to community acceptance:

## Conservative by Nature

The kernel community naturally resists dependencies and complexity. Simple, self-contained macros remove major objections to Rust adoption.

## Renewed Interest

Maintainers previously skeptical are taking a second look. Networking, filesystem, and driver subsystems are reconsidering Rust.

## Production Ready

These improvements help make Rust for Linux feel more production-ready for broader kernel usage, with fewer limitations

# Call to Action: How You Can Help

01

---

## Test These Features

They're in nightly Rust right now. Enable feature flags, convert procedural macros, find edge cases, report results.

03

---

## Document Experiences

Write blog posts about conversions, share build time improvements, show concrete examples of workflow benefits.

02

---

## Provide Feedback

The Rust team needs to know what patterns are common in kernel code, what syntax is awkward, what error messages confuse.

04

---

## Help with Adoption

Create migration guides, update documentation, teach kernel developers effective usage patterns.

# Getting Started Today

You can begin experimenting with these features immediately:

```
#![feature(macro_attr)]  
#![feature(macro_derive)]
```

```
// Now you can write declarative attribute and derive macros
```

## Where to Contribute

- File issues on the Rust repository
- Participate in discussions on Zulip
- Engage in internals forum conversations
- Test real-world conversion scenarios



### Think Bigger

What other pain points in kernel development could be addressed? What patterns do we use repeatedly? How can we make Rust not just possible but pleasant for kernel development?

# Thank You!

## Questions & Discussion

---

### Contact Information:

Vincenzo Palazzo

[vincenzopalazzodev@gmail.com](mailto:vincenzopalazzodev@gmail.com)

### Referenced RFCs:

- [RFC 3714: Future Fragment Fields Syntax](#)
- [Declarative Attribute Macros RFC](#)

---

### Your Call to Action:

Try these powerful new features on **nightly Rust** today!

Enable `#![feature(macro_attr)]` and `#![feature(macro_derive)]` in your projects to explore declarative attribute and derive macros.