Linux Rust Documentation

Release 5.4.10-custom

The kernel development community

Aug 21, 2023

CONTENTS

Documentation related to Rust within the kernel. To start using Rust in the kernel, please read the Quick Start guide.

CHAPTER

ONE

QUICK START

This document describes how to get started with kernel development in Rust.

1.1 Requirements: Building

This section explains how to fetch the tools needed for building.

Some of these requirements might be available from Linux distributions under names like `rustc`, `rust-src`, `rust-bindgen`, etc. However, at the time of writing, they are likely not to be recent enough unless the distribution tracks the latest releases.

To easily check whether the requirements are met, the following target can be used:

```
make LLVM=1 rustavailable
```

This triggers the same logic used by Kconfig to determine whether `RUST_IS_AVAILABLE` should be enabled; but it also explains why not if that is the case.

1.1.1 rustc

A particular version of the Rust compiler is required. Newer versions may or may not work because, for the moment, the kernel depends on some unstable Rust features.

If `rustup` is being used, enter the checked out source code directory and run:

```
rustup override set $(scripts/min-tool-version.sh rustc)
```

Otherwise, fetch a standalone installer or install `rustup` from:

> https://www.rust-lang.org

1.1.2 Rust standard library source

The Rust standard library source is required because the build system will cross-compile `core` and `alloc`.

If `rustup` is being used, run:

```
rustup component add rust-src
```

The components are installed per toolchain, thus upgrading the Rust compiler version later on requires re-adding the component.

Otherwise, if a standalone installer is used, the Rust repository may be cloned into the installation folder of the toolchain:

```
git clone --recurse-submodules \
        --branch $(scripts/min-tool-version.sh rustc) \
        https://github.com/rust-lang/rust \
        $(rustc --print sysroot)/lib/rustlib/src/rust
```

In this case, upgrading the Rust compiler version later on requires manually updating this clone.

1.1.3 libclang

`libclang` (part of LLVM) is used by `bindgen` to understand the C code in the kernel, which means LLVM needs to be installed; like when the kernel is compiled with `CC=clang` or `LLVM=1`.

Linux distributions are likely to have a suitable one available, so it is best to check that first.

There are also some binaries for several systems and architectures uploaded at:

> https://releases.llvm.org/download.html

Otherwise, building LLVM takes quite a while, but it is not a complex process:

> https://llvm.org/docs/GettingStarted.html#getting-the-source-code-and-building-llvm

Please see Documentation/kbuild/llvm.rst for more information and further ways to fetch pre-built releases and distribution packages.

1.1.4 bindgen

The bindings to the C side of the kernel are generated at build time using the `bindgen` tool. A particular version is required.

Install it via (note that this will download and build the tool from source):

```
cargo install --locked --version $(scripts/min-tool-version.sh bindgen) bindgen
```

1.2 Requirements: Developing

This section explains how to fetch the tools needed for developing. That is, they are not needed when just building the kernel.

1.2.1 rustfmt

The `rustfmt` tool is used to automatically format all the Rust kernel code, including the generated C bindings (for details, please see Coding Guidelines).

If `rustup` is being used, its `default` profile already installs the tool, thus nothing needs to be done. If another profile is being used, the component can be installed manually:

```
rustup component add rustfmt
```

The standalone installers also come with `rustfmt`.

1.2.2 clippy

`clippy` is a Rust linter. Running it provides extra warnings for Rust code. It can be run by passing `CLIPPY=1` to `make` (for details, please see General Information).

If `rustup` is being used, its `default` profile already installs the tool, thus nothing needs to be done. If another profile is being used, the component can be installed manually:

```
rustup component add clippy
```

The standalone installers also come with `clippy`.

1.2.3 cargo

`cargo` is the Rust native build system. It is currently required to run the tests since it is used to build a custom standard library that contains the facilities provided by the custom `alloc` in the kernel. The tests can be run using the `rusttest` Make target.

If `rustup` is being used, all the profiles already install the tool, thus nothing needs to be done.

The standalone installers also come with `cargo`.

1.2.4 rustdoc

`rustdoc` is the documentation tool for Rust. It generates pretty HTML documentation for Rust code (for details, please see General Information).

`rustdoc` is also used to test the examples provided in documented Rust code (called doctests or documentation tests). The `rusttest` Make target uses this feature.

If `rustup` is being used, all the profiles already install the tool, thus nothing needs to be done.

The standalone installers also come with `rustdoc`.

1.2.5 rust-analyzer

The rust-analyzer language server can be used with many editors to enable syntax highlighting, completion, go to definition, and other features.

`rust-analyzer` needs a configuration file, `rust-project.json`, which can be generated by the `rust-analyzer` Make target.

1.3 Configuration

`Rust support` (`CONFIG_RUST`) needs to be enabled in the `General setup` menu. The option is only shown if a suitable Rust toolchain is found (see above), as long as the other requirements are met. In turn, this will make visible the rest of options that depend on Rust.

Afterwards, go to:

```
Kernel hacking
    -> Sample kernel code
        -> Rust samples
```

And enable some sample modules either as built-in or as loadable.

1.4 Building

Building a kernel with a complete LLVM toolchain is the best supported setup at the moment. That is:

```
make LLVM=1
```

For architectures that do not support a full LLVM toolchain, use:

```
make CC=clang
```

Using GCC also works for some configurations, but it is very experimental at the moment.

1.5 Hacking

To dive deeper, take a look at the source code of the samples at `samples/rust/`, the Rust support code under `rust/` and the `Rust hacking` menu under `Kernel hacking`.

If GDB/Binutils is used and Rust symbols are not getting demangled, the reason is the toolchain does not support Rust's new v0 mangling scheme yet. There are a few ways out:

- Install a newer release (GDB >= 10.2, Binutils >= 2.36).
- Some versions of GDB (e.g. vanilla GDB 10.1) are able to use the pre-demangled names embedded in the debug info (`CONFIG_DEBUG_INFO`).

CHAPTER

TWO

GENERAL INFORMATION

This document contains useful information to know when working with the Rust support in the kernel.

2.1 Code documentation

Rust kernel code is documented using `rustdoc`, its built-in documentation generator.

The generated HTML docs include integrated search, linked items (e.g. types, functions, constants), source code, etc. They may be read at (TODO: link when in mainline and generated alongside the rest of the documentation):

> http://kernel.org/

The docs can also be easily generated and read locally. This is quite fast (same order as compiling the code itself) and no special tools or environment are needed. This has the added advantage that they will be tailored to the particular kernel configuration used. To generate them, use the `rustdoc` target with the same invocation used for compilation, e.g.:

```
make LLVM=1 rustdoc
```

To read the docs locally in your web browser, run e.g.:

```
xdg-open rust/doc/kernel/index.html
```

To learn about how to write the documentation, please see Coding Guidelines.

2.2 Extra lints

While `rustc` is a very helpful compiler, some extra lints and analyses are available via `clippy`, a Rust linter. To enable it, pass `CLIPPY=1` to the same invocation used for compilation, e.g.:

```
make LLVM=1 CLIPPY=1
```

Please note that Clippy may change code generation, thus it should not be enabled while building a production kernel.

2.3 Abstractions vs. bindings

Abstractions are Rust code wrapping kernel functionality from the C side.

In order to use functions and types from the C side, bindings are created. Bindings are the declarations for Rust of those functions and types from the C side.

For instance, one may write a `Mutex` abstraction in Rust which wraps a `struct mutex` from the C side and calls its functions through the bindings.

Abstractions are not available for all the kernel internal APIs and concepts, but it is intended that coverage is expanded as time goes on. ”Leaf” modules (e.g. drivers) should not use the C bindings directly. Instead, subsystems should provide as-safe-as-possible abstractions as needed.

2.4 Conditional compilation

Rust code has access to conditional compilation based on the kernel configuration:

```
#[cfg(CONFIG_X)]       // Enabled               (`y` or `m`)
#[cfg(CONFIG_X="y")]   // Enabled as a built-in (`y`)
#[cfg(CONFIG_X="m")]   // Enabled as a module   (`m`)
#[cfg(not(CONFIG_X))]  // Disabled
```

CHAPTER

THREE

CODING GUIDELINES

This document describes how to write Rust code in the kernel.

3.1 Style & formatting

The code should be formatted using `rustfmt`. In this way, a person contributing from time to time to the kernel does not need to learn and remember one more style guide. More importantly, reviewers and maintainers do not need to spend time pointing out style issues anymore, and thus less patch roundtrips may be needed to land a change.

Note: Conventions on comments and documentation are not checked by `rustfmt`. Thus those are still needed to be taken care of.

The default settings of `rustfmt` are used. This means the idiomatic Rust style is followed. For instance, 4 spaces are used for indentation rather than tabs.

It is convenient to instruct editors/IDEs to format while typing, when saving or at commit time. However, if for some reason reformatting the entire kernel Rust sources is needed at some point, the following can be run:

```
make LLVM=1 rustfmt
```

It is also possible to check if everything is formatted (printing a diff otherwise), for instance for a CI, with:

```
make LLVM=1 rustfmtcheck
```

Like `clang-format` for the rest of the kernel, `rustfmt` works on individual files, and does not require a kernel configuration. Sometimes it may even work with broken code.

3.2 Comments

"Normal" comments (i.e. `//`, rather than code documentation which starts with `///` or `//!`) are written in Markdown the same way as documentation comments are, even though they will not be rendered. This improves consistency, simplifies the rules and allows to move content between the two kinds of comments more easily. For instance:

```
// `object` is ready to be handled now.
f(object);
```

Furthermore, just like documentation, comments are capitalized at the beginning of a sentence and ended with a period (even if it is a single sentence). This includes `// SAFETY:`, `// TODO:` and other "tagged" comments, e.g.:

```
// FIXME: The error should be handled properly.
```

Comments should not be used for documentation purposes: comments are intended for implementation details, not users. This distinction is useful even if the reader of the source file is both an implementor and a user of an API. In fact, sometimes it is useful to use both comments and documentation at the same time. For instance, for a `TODO` list or to comment on the documentation itself. For the latter case, comments can be inserted in the middle; that is, closer to the line of documentation to be commented. For any other case, comments are written after the documentation, e.g.:

```
/// Returns a new [`Foo`].
///
/// # Examples
///
// TODO: Find a better example.
/// ```
/// let foo = f(42);
/// ```
// FIXME: Use fallible approach.
pub fn f(x: i32) -> Foo {
    // ...
}
```

One special kind of comments are the `// SAFETY:` comments. These must appear before every `unsafe` block, and they explain why the code inside the block is correct/sound, i.e. why it cannot trigger undefined behavior in any case, e.g.:

```
// SAFETY: `p` is valid by the safety requirements.
unsafe { *p = 0; }
```

`// SAFETY:` comments are not to be confused with the `# Safety` sections in code documentation. `# Safety` sections specify the contract that callers (for functions) or implementors (for traits) need to abide by. `// SAFETY:` comments show why a call (for functions) or implementation (for traits) actually respects the preconditions stated in a `# Safety` section or the language reference.

3.3 Code documentation

Rust kernel code is not documented like C kernel code (i.e. via kernel-doc). Instead, the usual system for documenting Rust code is used: the `rustdoc` tool, which uses Markdown (a lightweight markup language).

To learn Markdown, there are many guides available out there. For instance, the one at:

> https://commonmark.org/help/

This is how a well-documented Rust function may look like:

```
/// Returns the contained [`Some`] value, consuming the `self` value,
/// without checking that the value is not [`None`].
///
/// # Safety
///
/// Calling this method on [`None`] is *[undefined behavior]*.
///
/// [undefined behavior]: https://doc.rust-lang.org/reference/behavior-considered-
↪undefined.html
///
/// # Examples
///
/// ```
/// let x = Some("air");
/// assert_eq!(unsafe { x.unwrap_unchecked() }, "air");
/// ```
pub unsafe fn unwrap_unchecked(self) -> T {
    match self {
        Some(val) => val,

        // SAFETY: The safety contract must be upheld by the caller.
        None => unsafe { hint::unreachable_unchecked() },
    }
}
```

This example showcases a few `rustdoc` features and some conventions followed in the kernel:

- The first paragraph must be a single sentence briefly describing what the documented item does. Further explanations must go in extra paragraphs.
- Unsafe functions must document their safety preconditions under a `# Safety` section.
- While not shown here, if a function may panic, the conditions under which that happens must be described under a `# Panics` section.

  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`.
- If providing examples of usage would help readers, they must be written in a section called `# Examples`.
- Rust items (functions, types, constants…) must be linked appropriately (`rustdoc` will create a link automatically).
- Any `unsafe` block must be preceded by a `// SAFETY:` comment describing why the code inside is sound.

> While sometimes the reason might look trivial and therefore unneeded, writing these comments is not just a good way of documenting what has been taken into account, but most importantly, it provides a way to know that there are no extra implicit constraints.

To learn more about how to write documentation for Rust and extra features, please take a look at the `rustdoc` book at:

> https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html

3.4 Naming

Rust kernel code follows the usual Rust naming conventions:

> https://rust-lang.github.io/api-guidelines/naming.html

When existing C concepts (e.g. macros, functions, objects...) are wrapped into a Rust abstraction, a name as close as reasonably possible to the C side should be used in order to avoid confusion and to improve readability when switching back and forth between the C and Rust sides. For instance, macros such as `pr_info` from C are named the same in the Rust side.

Having said that, casing should be adjusted to follow the Rust naming conventions, and namespacing introduced by modules and types should not be repeated in the item names. For instance, when wrapping constants like:

```
#define GPIO_LINE_DIRECTION_IN  0
#define GPIO_LINE_DIRECTION_OUT 1
```

The equivalent in Rust may look like (ignoring documentation):

```
pub mod gpio {
    pub enum LineDirection {
        In = bindings::GPIO_LINE_DIRECTION_IN as _,
        Out = bindings::GPIO_LINE_DIRECTION_OUT as _,
    }
}
```

That is, the equivalent of `GPIO_LINE_DIRECTION_IN` would be referred to as `gpio::LineDirection::In`. In particular, it should not be named `gpio::gpio_line_direction::GPIO_LINE_DIRECTION_IN`.

CHAPTER

FOUR

ARCH SUPPORT

Currently, the Rust compiler (`rustc`) uses LLVM for code generation, which limits the supported architectures that can be targeted. In addition, support for building the kernel with LLVM/Clang varies (please see Documentation/kbuild/llvm.rst). This support is needed for `bindgen` which uses `libclang`.

Below is a general summary of architectures that currently work. Level of support corresponds to `S` values in the `MAINTAINERS` file.

Architecture	Level of support	Constraints
`um`	Maintained	`x86_64` only.
`x86`	Maintained	`x86_64` only.