The Rust toolchain in the kernel

Miguel Ojeda

ojeda@kernel.org

Which particular Rust toolchain is needed?

What is `RUSTC_BOOTSTRAP`?

Why do we need it?

Which components are required
to build, test, document…?

Compiler (`rustc`)

Standard library source (`rust-src`)

Bindings generator (`bindgen`)

Documentation generator (`rustdoc`)

Linter (`clippy`)

Formatter (`rustfmt`)

Build system (`cargo`)

Standard library binaries (`rust-std`)

Why is a bindings generator required?

Could you have the generated version of the bindings in-tree?

Which parts of the standard library are required?

Do they need to be compiled in a particular way?

Which version of LLVM `rustc` requires?

How should distributions provide this toolchain?

Should it be a separate one from the main Rust packages they may otherwise have?

Should we provide pre-compiled toolchains

from kernel.org?

Which architectures are supported so far?

Which ones may be soon supported?

Supported architectures

`arm` (`armv6` only)

`arm64`

`powerpc` (`ppc64le` only)

`riscv` (`riscv64` only)

`x86` (`x86_64` only)

See `Documentation/rust/arch-support.rst`

Supported architectures

`arm` (`armv6` only)

`arm64`

`powerpc` (`ppc64le` only)

`riscv` (`riscv64` only)

`x86` (`x86_64` only)

...so far!

32-bit and other restrictions should be easy to remove

Kernel LLVM builds work for `mips` and `s390`

GCC codegen paths should open up more

See `Documentation/rust/arch-support.rst`

Are there alternative Rust compilers?

How advanced they are?

Rust codegen paths for the kernel

rustc_codegen_gcc	rustc_codegen_llvm	Rust GCC
Already passes most rustc tests	*Main one*	*Expected in 1-2 years (rough estimate)*

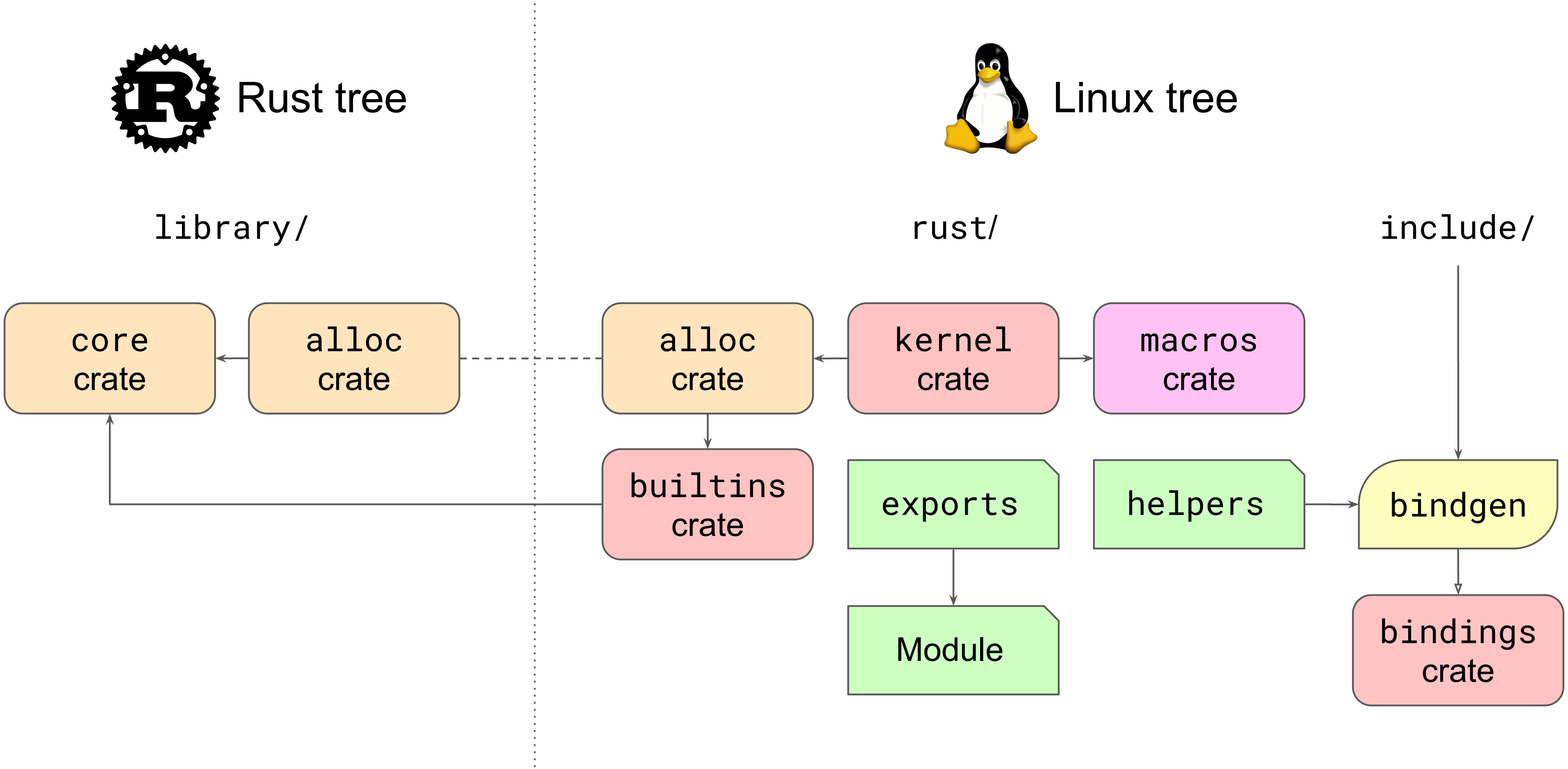
...?

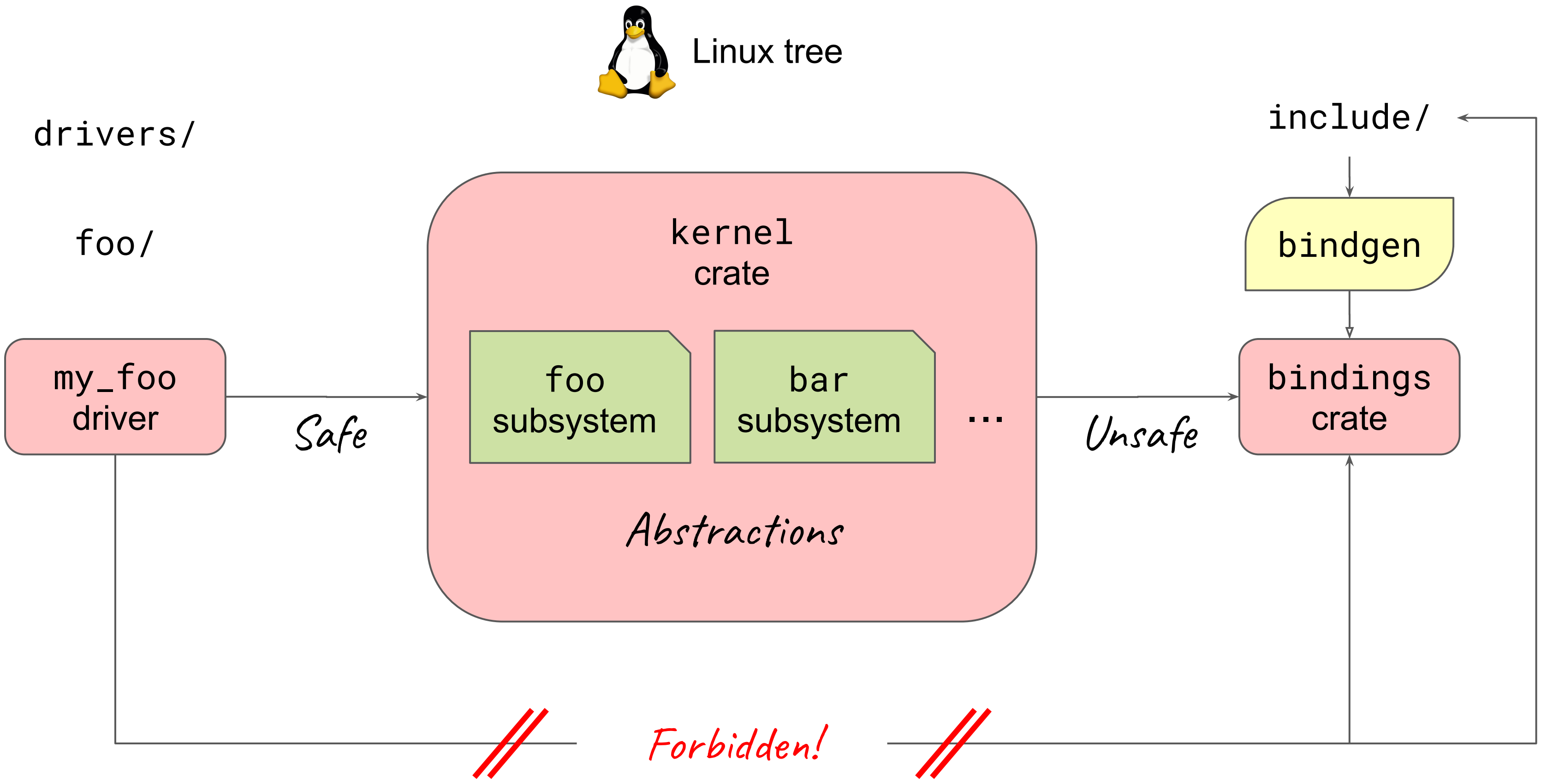
The Rust toolchain in the kernel

Miguel Ojeda

ojeda@kernel.org

Backup slides


Rust tree
Linux tree
library/
rust/
include/
core crate
alloc crate
alloc crate
kernel crate
macros crate
builtins crate
exports
helpers
bindgen
Module
bindings crate



Linux tree
drivers/
foo/
my_foo driver
Safe
kernel crate
foo subsystem
bar subsystem
...
Abstractions
Unsafe
include/
bindgen
bindings crate
Forbidden!

What else does Rust offer?

Documentation generator

Unit & integration tests

Static analyzer

C ↔ Rust bindings generators

Linter

Tooling

Macro debugging

Formatter

IDE tooling

Great compiler error messages

UBSAN-like interpreter

plus the usual friends: `gdb`, `lldb`, `perf`, `valgrind`...

GCC

```
$ aarch64-linux-gnu-
aarch64-linux-gnu-addr2line     aarch64-linux-gnu-gcc-7
aarch64-linux-gnu-ar            aarch64-linux-gnu-gcc-ar
aarch64-linux-gnu-as            aarch64-linux-gnu-gcc-ar-7
aarch64-linux-gnu-c++filt       aarch64-linux-gnu-gcc-nm
aarch64-linux-gnu-cpp           aarch64-linux-gnu-gcc-nm-7
aarch64-linux-gnu-cpp-7         aarch64-linux-gnu-gcc-ranlib
aarch64-linux-gnu-dwp           aarch64-linux-gnu-gcc-ranlib-7
aarch64-linux-gnu-elfedit       aarch64-linux-gnu-gcov
aarch64-linux-gnu-gcc           aarch64-linux-gnu-gcov-7
```

```
-fomit-frame-pointer
-ftrapv

...

-mno-red-zone
-mcmodel=kernel

...

-freg-struct-return
-fpack-struct
-mregparm=num
```

Clang

General Cross-Compilation Options in Clang

Target Triple

The basic option is to define the target architecture. For that, use `-target <triple>`. If you don't specify the target, CPU names won't match (since Clang assumes the host triple), and the compilation will go ahead, creating code for the host platform, which will break later on when assembling or linking.

The triple has the general format `<arch><sub>-<vendor>-<sys>-<abi>`, where:

- `arch` = `x86_64`, `i386`, `arm`, `thumb`, `mips`, etc.
- `sub` = for ex. on ARM: `v5`, `v6m`, `v7a`, `v7m`, etc.
- `vendor` = `pc`, `apple`, `nvidia`, `ibm`, etc.
- `sys` = `none`, `linux`, `win32`, `darwin`, `cuda`, etc.
- `abi` = `eabi`, `gnu`, `android`, `macho`, `elf`, etc.

The sub-architecture options are available for their own architectures, of course, so "x86v7a" doesn't make sense. The vendor needs to be specified only if there's a relevant change, for instance between PC and Apple. Most of the time it can be omitted (and Unknown) will be assumed, which sets the defaults for the specified architecture. The system name is generally the OS (linux, darwin), but could be special like the bare-metal "none".

When a parameter is not important, it can be omitted, or you can choose `unknown` and the defaults will be used. If you choose a parameter that Clang doesn't know, like `blerg`, it'll ignore and assume `unknown`, which is not always desired, so be careful.

Finally, the ABI option is something that will pick default CPU/FPU, define the specific behaviour of your code (PCS, extensions), and also choose the correct library calls, etc.

```
-fomit-frame-pointer
-ftrapv

...

-mno-red-zone
-mcmodel=kernel

...

-freg-struct-return
-fpack-struct
-mregparm=num
```

rustc

Tier 1 with Host Tools

Tier 1 targets can be thought of as "guaranteed to work". The Rust project builds official binary releases for each tier 1 target, and automated testing ensures that each tier 1 target builds and passes tests after each change.

Tier 1 targets with host tools additionally support running tools like `rustc` and `cargo` natively on the target, and automated testing ensures that tests pass for the host tools as well. This allows the target to be used as a development platform, not just a compilation target. For the full requirements, see Tier 1 with Host Tools in the Target Tier Policy.

All tier 1 targets with host tools support the full standard library.

target	notes
`aarch64-unknown-linux-gnu`	ARM64 Linux (kernel 4.2, glibc 2.17+) [1]
`i686-pc-windows-gnu`	32-bit MinGW (Windows 7+)
`i686-pc-windows-msvc`	32-bit MSVC (Windows 7+)
`i686-unknown-linux-gnu`	32-bit Linux (kernel 2.6.32+, glibc 2.11+)
`x86_64-apple-darwin`	64-bit macOS (10.7+, Lion+)
`x86_64-pc-windows-gnu`	64-bit MinGW (Windows 7+)
`x86_64-pc-windows-msvc`	64-bit MSVC (Windows 7+)
`x86_64-unknown-linux-gnu`	64-bit Linux (kernel 2.6.32+, glibc 2.11+)

```json
{
    "arch": "x86_64",
    "code-model": "kernel",
    "cpu": "x86-64",
    "data-layout": "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:...",
    "disable-redzone": true,
    "eliminate-frame-pointer": false,
    "emit-debug-gdb-scripts": false,
    "env": "gnu",
    "features": "-mmx,-sse,-sse2,-sse3,+soft-float",
    "linker-flavor": "gcc",
    "linker-is-gnu": true,
    "llvm-target": "x86_64-elf",
    "max-atomic-width": 64,
    "os": "none",
    "panic-strategy": "abort",
     ...
}
```

```
-Cpanic=abort
-Cno-redzone
-Cllvm-args=...
-Clink-args=...

...
```

Handling GCC, Clang and rustc at the same time

Generating the target rustc file via Makefile or some script

Generate a description via Makefile or some script, then transform

Getting compiler to accept that description format

…?

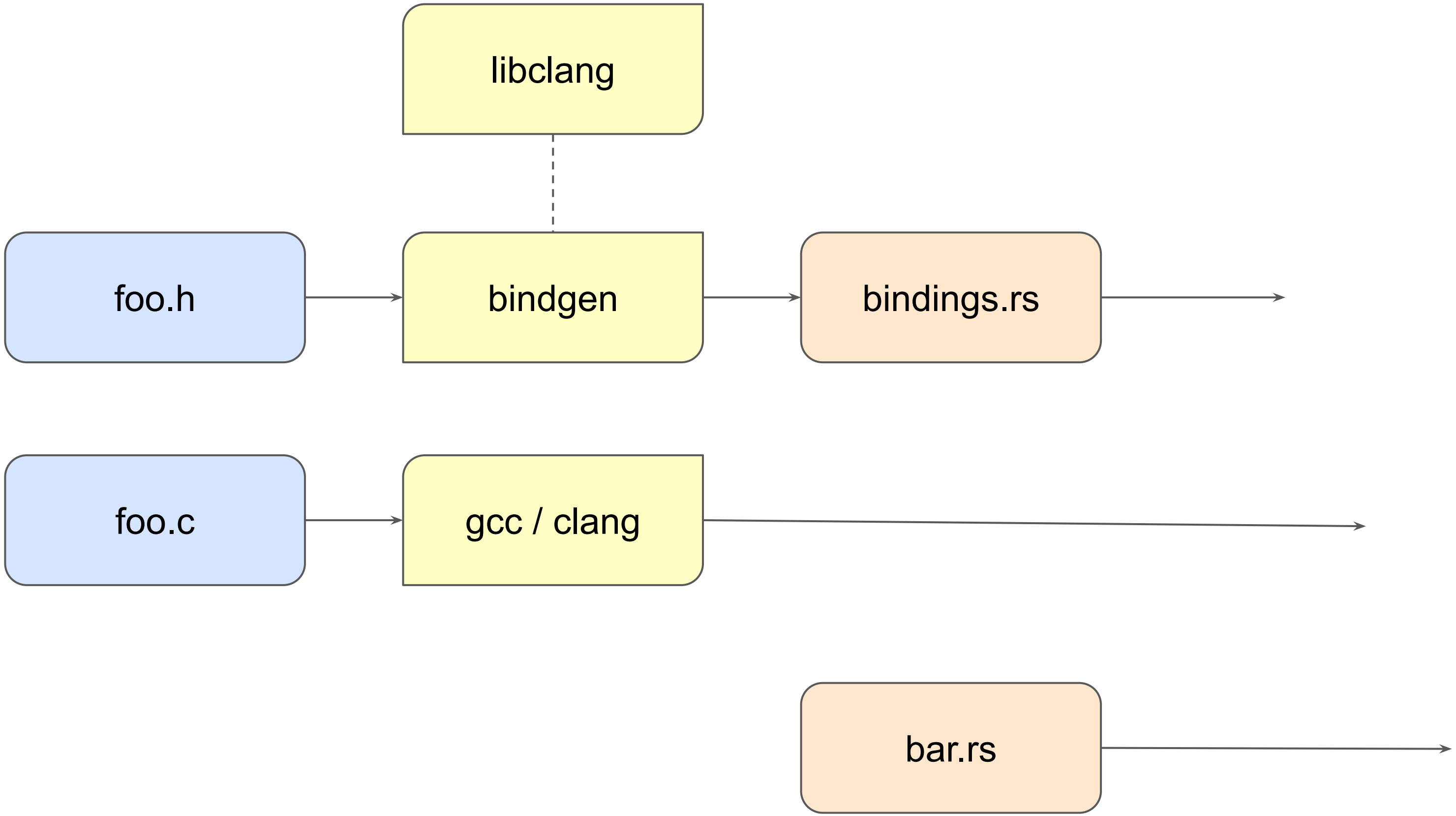
bindgen

“automatically generates
Rust FFI bindings to C
(and some C++) libraries”

```rust
/// A safe wrapper for `f`.
///
/// # Safety
///
/// Any preconditions required to guarantee no UB.
fn f_abstraction() -> i32 {
    unsafe { bindings::f() }
}

fn main() {
    println!("{}", f_abstraction());
}
```

Bindings

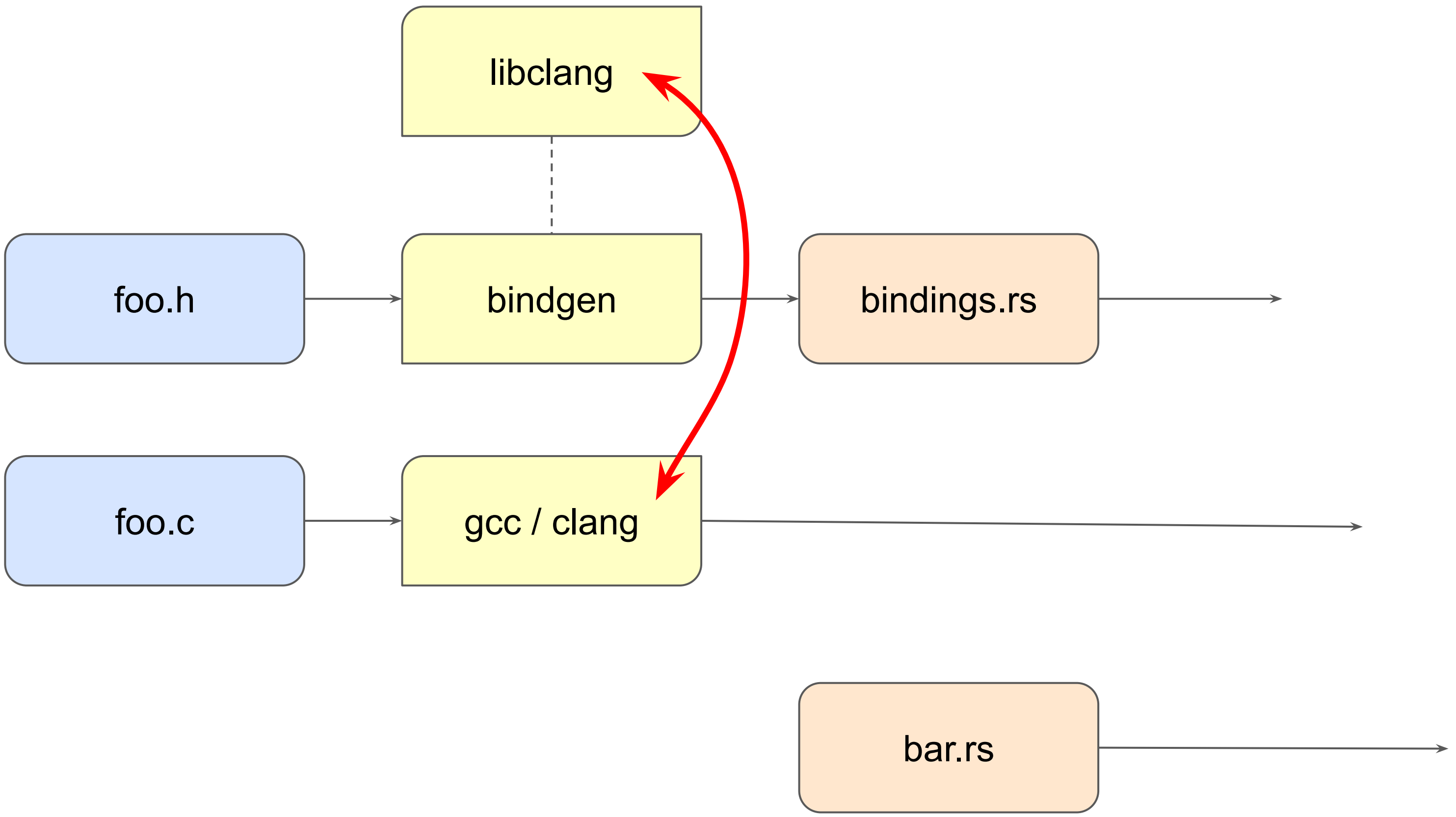


libclang
foo.h
bindgen
bindings.rs
foo.c
gcc / clang
bar.rs

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct rcu_cblist {
    pub head: *mut callback_head,
    pub tail: *mut *mut callback_head,
    pub len: c_types::c_long,
}
```

```
#[test]
fn bindgen_test_layout_rcu_cblist () {
    assert_eq!(
        ::core::mem::size_of::<rcu_cblist>(),
        24usize,
        concat!("Size of: ", stringify!(rcu_cblist))
    );
```

```
pub const ENERGY_PERF_BIAS_PERFORMANCE : u32 = 0;
pub const ENERGY_PERF_BIAS_BALANCE_PERFORMANCE : u32 = 4;
pub const ENERGY_PERF_BIAS_NORMAL : u32 = 6;
pub const ENERGY_PERF_BIAS_BALANCE_POWERSAVE : u32 = 8;
```


libclang
foo.h
bindgen
bindings.rs
foo.c
gcc / clang
bar.rs

4 Open ✓ 0 Closed

Support `_Noreturn`, `[[noreturn]]`, `__attribute__((noreturn))`

#2094 opened 24 days ago by ojeda

Support `unsafe_op_in_unsafe_fn` enhancement help wanted

#2063 opened on Jun 4 by ojeda

C javadoc comments are not Markdown-escaped, triggering `rustdoc` warnings

#2057 opened on May 29 by ojeda

Support for a GCC-based backend enhancement

#1949 opened on Dec 20, 2020 by ojeda

```c
#define __div_x64(dividend, divisor) ({                              \
        BUILD_BUG_ON_MSG(sizeof(divisor) > sizeof(u32),             \
                         "prefer __div64_x64");                     \
        __builtin_choose_expr(                                      \
                is_signed_type(typeof(dividend)),                   \
                div_s64((dividend), (divisor)),                     \
                div_u64((dividend), (divisor)));                    \
})

#define __div_64(dividend, divisor)                                 \
        _Generic((divisor),                                         \
        s64: __div64_x64((dividend), (divisor)),                    \
        u64: __div64_x64((dividend), (divisor)),                    \
        default: __div_x64((dividend), (divisor)))
```

```
__noreturn void rust_helper_BUG(void)
{
    BUG();
}
```

```rust
#[test]
#[host]
fn test_that_runs_in_the_host() {
    // Something that can be tested in the host.
}

#[test]
#[user]
fn test_that_runs_in_the_target's_userspace() {
    // Something that must be tested in the target,
    // but the test runs in userspace.
}

#[test]
#[kernel]
fn test_that_runs_in_the_target's_kernelspace() {
    // Something that must be tested in the target,
    // but the test runs in kernelspace.
}
```

Crate std

1.0.0 [-][src]

[-] ## The Rust Standard Library

The Rust Standard Library is the foundation of portable Rust software, a set of minimal and battle-tested shared abstractions for the broader Rust ecosystem. It offers core types, like `Vec<T>` and `Option<T>`, library-defined operations on language primitives, standard macros, I/O and multithreading, among many other things.

`std` is available to all Rust crates by default. Therefore, the standard library can be accessed in `use` statements through the path `std`, as in `use std::env`.

How to read this documentation

If you already know the name of what you are looking for, the fastest way to find it is to use the search bar at the top of the page.

Otherwise, you may want to jump to one of these useful sections:

- `std::*` modules
- Primitive types
- Standard macros
- The Rust Prelude

If this is your first time, the documentation for the standard library is written to be casually perused. Clicking on interesting things should generally lead you to interesting places. Still, there are important bits you don't want to miss, so read on for a tour of the standard library and its documentation!

Once you are familiar with the contents of the standard library you may begin to find the verbosity of the prose distracting. At this stage in your development you may want to press the `[-]` button near the top of the page to collapse it into a more skimmable

Crate kernel

[-][src]

[-] The `kernel` crate.

This crate contains the kernel APIs that have been ported or wrapped for usage by Rust code in the kernel and is shared by all of them.

In other words, all the rest of the Rust code in the kernel (e.g. kernel modules written in Rust) depends on `core`, `alloc` and this crate.

If you need a kernel C API that is not ported or wrapped yet here, then do so first instead of bypassing this crate.

Modules

buffer	Struct for writing to a pre-allocated buffer with the `write!` macro.
c_types	C types for the bindings.
chrdev	Character devices.
file	Files and file descriptors.
file_operations	File operations.
io_buffer	Buffers used in IO.
iov_iter	IO vector iterators.
linked_list	Linked lists.
miscdev	Miscellaneous devices.
of	Devicetree and Open Firmware abstractions.
pages	Kernel page allocation and management.
platdev	Platform devices.
prelude	The `kernel` prelude.
print	Printing facilities.

Conditional compilation

Rust code has access to conditional compilation based on the kernel config

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

Coding guidelines

No direct access to C bindings

No undocumented public APIs

No implicit `unsafe` block

Docs follows Rust standard library style

`// SAFETY` proofs for all `unsafe` blocks

Clippy linting enabled

Automatic formatting enforced

Rust 2018 edition & idioms

No unneeded panics

No infallible allocations

...

Coding guidelines

No direct access to C bindings

No undocumented public APIs

No implicit `unsafe` block

Docs follows Rust standard library style

`// SAFETY` proofs for all `unsafe` blocks

Clippy linting enabled

Automatic formatting enforced

Rust 2018 edition & idioms

No unneeded panics

No infallible allocations

...

Aiming to be as strict as possible