Effective in 2026, to align with our trunk stable development model and ensure platform stability for the ecosystem, we will publish source code to AOSP in Q2 and Q4. For building and contributing to AOSP, use `android-latest-release`. The `android-latest-release` manifest branch will always reference the most recent release pushed to AOSP. For more information, see Changes to AOSP (/docs/whatsnew/site-updates#aosp-changes).

Bindgen binding modules

The build system supports generating bindgen bindings through the `rust_bindgen` module type. Bindgen provides Rust FFI bindings to C libraries (with some limited C++ support, which requires setting the `cppstd` property).

Basic rust_bindgen usage

What follows is an example of how to define a module that uses bindgen, and how to use that module as a crate. If you need to use bindgen bindings through an `include!()` macro, such as for external code, see Source generators (/docs/setup/build/rust/building-rust-modules/source-code-generators/source-code-gen-intro).

Example C library to call from Rust

An example C library which defines a struct and function for use in Rust follows.

`external/rust/libbuzz/libbuzz.h`

```
typedef struct foo {
    int x;
} foo;

void fizz(int i, foo* cs);
```

`external/rust/libbuzz/libbuzz.c`

```
#include <stdio.h>
#include "libbuzz.h"

void fizz(int i, foo* my_foo){
    printf("hello from c! i = %i, my_foo->x = %i\n", i, my_foo->x);
}
```

Define a rust_bindgen module

Define a wrapper header, `external/rust/libbuzz/libbuzz_wrapper.h`, which includes all relevant headers:

```
// Include headers that are required for generating bindings in a wrapper header.
#include "libbuzz.h"
```

Important: By default the build system assumes the header is a C header unless the extension is **`.hpp`** or the **`cpp_std`** value is set. This determines which **`-std`** flag is passed to Clang. For additional details, see Notable rust_bindgen properties (#notable-rust-bindgen-properties).

Define the `Android.bp` file as `external/rust/libbuzz/Android.bp`:

```
cc_library {
    name: "libbuzz",
    srcs: ["libbuzz.c"],
}

rust_bindgen {
     name: "libbuzz_bindgen",

     // Crate name that's used to generate the rust_library variants.
     crate_name: "buzz_bindgen",

     // Path to the wrapper source file.
```

```
    wrapper_src: "libbuzz_wrapper.h",

    // 'source_stem' controls the output filename.
    // This is the filename that's used in an include! macro.
    //
    // In this case, we just use "bindings", which produces
    // "bindings.rs".
    source_stem: "bindings",

    // Bindgen-specific flags and options to customize the bindings.
    // See the bindgen manual for more information.
    bindgen_flags: ["--verbose"],

    // Clang flags to be used when generating the bindings.
    cflags: ["-DSOME_FLAG"],

    // Shared, static, and header libraries which export the necessary
    // include directories must be specified.
    //
    // These libraries will also be included in the crate if static,
    // or propagated to dependents if shared.
    // static_libs: ["libbuzz"]
    // header_libs: ["libbuzz"]
    shared_libs: ["libbuzz"],
}
```

To learn more about using bindgen flags, see the bindgen manual section on Customizing the Generated Bindings (https://rust-lang.github.io/rust-bindgen/customizing-generated-bindings.html).

If you used this section to define a `rust_bindgen` module as a prerequisite to using the `include!()` macro, return to Prerequisite (/docs/setup/build/rust/building-rust-modules/source-code-generators/source-code-gen-intro#prerequisite) on the Source Generators page. If not, proceed with the next sections.

Use bindings as a crate

Create `external/rust/hello_bindgen/Android.bp` with the following content:

```
rust_binary {
   name: "hello_bindgen",
   srcs: ["main.rs"],
```

```
    // Add the rust_bindgen module as if it were a rust_library dependency.
    rustlibs: ["libbuzz_bindgen"],
}
```

Create `external/rust/hello_bindgen/src/main.rs` with the following content:

```
//! Example crate for testing bindgen bindings

fn main() {
    let mut x = buzz_bindgen::foo { x: 2 };
    unsafe { buzz_bindgen::fizz(1, &mut x as *mut buzz_bindgen::foo) }
}
```

Then, call `m hello_bindgen` to build the binary.

Note: Using bindgen bindings as a crate is the preferred method for platform code to leverage bindings. To see an example of providing bindings for use with the `include!()` macro, typically for code under `external/`, see Use include!() to include generated source (/docs/setup/build/rust/building-rust-modules/source-code-generators/source-code-gen-intro#using-include-for-generated-source-inclusion)
.

Test bindgen bindings

Bindgen bindings typically contain a number of generated layout tests to prevent memory layout mismatches. AOSP recommends that you have a test module defined for these tests, and that the tests run as part of your project's normal test suite.

You can produce a test binary for these by defining a `rust_test` module in `external/rust/hello_bindgen/Android.bp`:

```
rust_test {
    name: "bindings_test",
    srcs: [
        ":libbuzz_bindgen",
```

```
    ],
    crate_name: "buzz_bindings_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,

    // Be sure to disable lints as the generated source
    // is not guaranteed to be lint-free.
    clippy_lints: "none",
    lints: "none",
}
```

Visibility and linkage

Generated bindings are usually small, as they consist of type definitions, function signatures, and related constants. As a result, it's wasteful to link these libraries dynamically. We disabled dynamic linkage for these modules so that using them with `rustlibs` automatically selects a static variant.

By default, `rust_bindgen` modules have a `visibility` property of `[":__subpackages__"]`, which permits modules in the same `Android.bp` file or those beneath it in the directory hierarchy to see it. This serves two purposes:

- Discourages the use of raw C bindings elsewhere in the tree.
- Avoids diamond linking issues with a mix of static and dynamic linkage.

You should provide a safe wrapper library around the generated module you added in the same directory tree as the bindings, which is intended for other developers to use. If this doesn't work for your use case, you can add additional packages to `visibility` (https://android.googlesource.com/platform/build/soong/+/refs/heads/android17-release/README.md#visibility). When adding additional visibility scopes, don't add two scopes that might be linked into the same process in the future, as this might fail to link.

Notable rust_bindgen properties

The properties defined in this section are in addition to the Important common properties (/docs/setup/build/rust/building-rust-modules/android-rust-modules#important-common-properties) that apply to all modules. These are either important to Rust bindgen modules, or exhibit unique behavior specific to the `rust_bindgen` module type.

stem, name, crate_name

`rust_bindgen` produces library variants, so they share the same requirements with the `rust_library` modules for the `stem`, `name`, and `crate_name` properties. For reference, see Notable Rust library properties (/docs/setup/build/rust/building-rust-modules/library-modules).

wrapper_src

This is the relative path to a wrapper header file that includes the headers required for these bindings. The file extension determines how to interpret the header and determines which `-std` flag to use by default. This is assumed to be a C header unless the extension is either `.hh` or `.hpp`. If your C++ header must have some other extension, set the `cpp_std` property to override the default behavior that assumes the file is a C file.

source_stem

This is the filename for the *generated source file*. This field *must* be defined, even if you're using the bindings as a crate, since the `stem` property only controls the output filename for the generated library variants. If a module depends on multiple source generators (such as `bindgen` and `protobuf`) as source rather than as crates through `rustlibs`, ensure that all source generators that are dependencies of that module have unique `source_stem` values. Dependent modules copy sources from all `SourceProvider` dependencies which are defined in `srcs` to a common `OUT_DIR` directory, so collisions in `source_stem` would result in the generated source files being overwritten in the `OUT_DIR` directory.

c_std

This is a string representing which C-standard version to use. Valid values are listed here:

- A specific version, such as `gnu11`
- `experimental`, which is a value defined by the build system in `build/soong/cc/config/global.go`, can use draft versions like C++1z (https://android.googlesource.com/platform/build/soong/+/393bffee78a8c825d6bc64bf096b6df6978f7150/cc/config/global.go) when they're available
- Unset or `""`, which indicates that the build system default should be used

If this is set, the file extension is ignored and the header is assumed to be a C header. This can't be set at the same time as `cpp_std`.

cpp_std

`cpp_std` is a string representing which C standard version to use. Valid values are listed here:

- A specific version, such as `gnu++11`
- `experimental`, which is a value defined by the build system in `build/soong/cc/config/global.go`, can use draft versions like C++1z (https://android.googlesource.com/platform/build/soong/+/393bffee78a8c825d6bc64bf096b6df6978f7150/cc/config/global.go) when they're available
- Unset or `""`, which indicates that the build system default should be used

If this is set, the file extension is ignored and the header is assumed to be a C++ header. This can't be set at the same time as `c_std`.

cflags

`cflags` provides a string list of Clang flags required to correctly interpret the headers.

custom_bindgen

For advanced use cases, bindgen can be used as a library, providing an API that can be manipulated as part of a custom Rust binary. The `custom_bindgen` field takes the module name of a `rust_binary_host` module, which uses the bindgen API instead of the normal `bindgen` binary.

This custom binary must expect arguments in a similar fashion to `bindgen`, such as

```
$ my_bindgen [flags] wrapper_header.h -o [output_path] -- [clang flags]
```

Most of this is handled by the `bindgen` library itself. To see an example of this usage, visit external/rust/crates/libsqlite3-sys/android/build.rs (https://android.googlesource.com/platform/external/rust/crates/libsqlite3-sys/+/19c12c3ecab39ec463ef04bafacc01cd87382c98/android/build.rs)
.

Additionally, the full set of library properties is available to control the compilation of the library, although these rarely need defining or changing.

handle_static_inline and static_inline_library

These two properties are meant to be used together and allow production of wrappers for static inline functions which can be included in the exported bindgen bindings.

To use them, set `handle_static_inline: true` and set `static_inline_library` to a corresponding `cc_library_static` which defines the `rust_bindgen` module as source input.

Example usage:

```
rust_bindgen {
    name: "libbindgen",
    wrapper_src: "src/any.h",
    crate_name: "bindgen",
    stem: "libbindgen",
    source_stem: "bindings",

    // Produce bindings for static inline functions
    handle_static_inline: true,
    static_inline_library: "libbindgen_staticfns"
}

cc_library_static {
    name: "libbindgen_staticfns",

    // This is the rust_bindgen module defined above
    srcs: [":libbindgen"],

    // The include path to the header file in the generated C file is
    // relative to build top.
    include_dirs: ["."],
}
```

Caution: If there are no static inline functions provided through the header file, then bindgen (as of 0.69.2) silently fails to output a C file, and the `cc_library_static` depending on the source fails compilation.

Last updated 2026-07-16 UTC.