x86_64-unknown-none

Tier: 3

Freestanding/bare-metal x86-64 binaries in ELF format: firmware, kernels, etc.

Target maintainers

- Harald Hoyer `harald@profian.com`, https://github.com/haraldh
- Mike Leany, https://github.com/mikeleany

Requirements

This target is cross-compiled. There is no support for `std`. There is no default allocator, but it's possible to use `alloc` by supplying an allocator.

By default, Rust code generated for this target does not use any vector or floating-point registers (e.g. SSE, AVX). This allows the generated code to run in environments, such as kernels, which may need to avoid the use of such registers or which may have special considerations about the use of such registers (e.g. saving and restoring them to avoid breaking userspace code using the same registers). You can change code generation to use additional CPU features via the `-C target-feature=` codegen options to rustc, or via the `#[target_feature]` mechanism within Rust code.

By default, code generated with this target should run on any `x86_64` hardware; enabling additional target features may raise this baseline.

Code generated with this target will use the `kernel` code model by default. You can change this using the `-C code-model=` option to rustc.

On `x86_64-unknown-none`, `extern "C"` uses the standard System V calling convention, without red zones.

This target generated binaries in the ELF format. Any alternate formats or special considerations for binary layout will require linker options or linker scripts.

Building the target

You can build Rust with support for the target by adding it to the `target` list in `config.toml`:

```toml
[build]
build-stage = 1
target = ["x86_64-unknown-none"]
```

Building Rust programs

Rust does not yet ship pre-compiled artifacts for this target. To compile for this target, you will either need to build Rust with the target enabled (see "Building the target" above), or build your own copy of `core` by using `build-std` or similar.

Testing

As `x86_64-unknown-none` supports a variety of different environments and does not support `std`, this target does not support running the Rust testsuite.

Cross-compilation toolchains and C code

If you want to compile C code along with Rust (such as for Rust crates with C dependencies), you will need an appropriate `x86_64` toolchain.

Rust *may* be able to use an `x86_64-linux-gnu-` toolchain with appropriate standalone flags to build for this toolchain (depending on the assumptions of that toolchain, see below), or you may wish to use a separate `x86_64-unknown-none` (or `x86_64-elf-`) toolchain.

On some `x86_64` hosts that use ELF binaries, you *may* be able to use the host C toolchain, if it does not introduce assumptions about the host environment that don't match the expectations of a standalone environment. Otherwise, you may need a separate toolchain for standalone/freestanding development, just as when cross-compiling from a non-`x86_64` platform.