

Cross compilation

Contents

- What do you need?
- Platforms
- What's a target platform?
- Determining the host platform config
- Choosing the host platform with Nix
- Specifying the host platform
- Cross compiling for the first time
- Real-world cross compiling of a Hello World example
- Developer environment with a cross compiler
- Next steps

Nixpkgs offers powerful tools to cross-compile software for various system types.

What do you need?

- Experience using C compilers
- Basic knowledge of the [Nix language](#)

Platforms

When compiling code, we can distinguish between the **build platform**, where the executable is *built*, and the **host platform**, where the compiled executable *runs*. ^[1]

Native compilation is the special case where those two platforms are the same. **Cross compilation** is the general case where those two platforms are not.

Cross compilation is needed when the host platform has limited resources (such as CPU) or when it's not easily accessible for development.

[Skip to main content](#)

The `nixpkgs` package collection has world-class support for cross compilation, after many years of hard work by the Nix community.

What's a target platform?

There is a third concept for a platform we call a **target platform**.

The target platform is relevant to cases where you want to build a compiler binary. In such cases, you would build a compiler on the *build platform*, run it to compile code on the *host platform*, and run the final executable on the *target platform*.

Since this is rarely needed, we will assume that the target is identical to the host.

Determining the host platform config

The build platform is determined automatically by Nix during the configure phase.

The host platform is best determined by running this command on the host platform:

```
$ $(nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 -A gnu-config)/config.guess  
aarch64-unknown-linux-gnu
```

In case this is not possible (for example, when the host platform is not easily accessible for development), the platform config has to be constructed manually via the following template:

```
<cpu>-<vendor>-<os>-<abi>
```

This string representation is used in `nixpkgs` for historic reasons.

Note that `<vendor>` is often `unknown` and `<abi>` is optional. There's also no unique identifier for a platform, for example `unknown` and `pc` are interchangeable (which is why the script is called `config.guess`).

If you can't install Nix, find a way to run `config.guess` (usually comes with the `autoconf` package) from the OS you're able to run on the host platform.

Some other common examples of platform configs:

[Skip to main content](#)

- aarch64-apple-darwin14
- aarch64-pc-linux-gnu
- x86_64-w64-mingw32
- aarch64-apple-ios

Note

macOS/Darwin is a special case, as not the whole OS is open-source. It's only possible to cross compile between `aarch64-darwin` and `x86_64-darwin`. `aarch64-darwin` support was recently added, so cross compilation is barely tested.

Choosing the host platform with Nix

`nixpkgs` comes with a set of predefined host platforms for cross compilation called `pkgsCross`.

It is possible to explore them in `nix repl`:

Note

Starting with Nix 2.19, `nix repl` requires the `-f` / `--file` flag:

```
$ nix repl -f '<nixpkgs>' -I nixpkgs=channel:nixos-23.11
```

```
$ nix repl '<nixpkgs>' -I nixpkgs=channel:nixos-23.11
Welcome to Nix 2.18.1. Type :? for help.
```

```
Loading '<nixpkgs>'...
Added 14200 variables.
```

```
nix-repl> pkgsCross.<TAB>
pkgsCross.aarch64-android          pkgsCross.musl-power
pkgsCross.aarch64-android-prebuilt pkgsCross.musl32
pkgsCross.aarch64-darwin           pkgsCross.musl64
pkgsCross.aarch64-embedded         pkgsCross.muslpi
pkgsCross.aarch64-multiplatform    pkgsCross.or1k
pkgsCross.aarch64-multiplatform-musl pkgsCross.pogoplug4
pkgsCross.aarch64be-embedded       pkgsCross.powernv
pkgsCross.amd64-netbsd             pkgsCross.ppc-embedded
pkgsCross.arm-embedded             pkgsCross.ppc64
pkgsCross.armhf-embedded           pkgsCross.ppc64-musl
pkgsCross.armv7a-android-prebuilt  pkgsCross.riscv-embedded
```

[Skip to main content](#)

```
pkgsCross.ben-nanonote
pkgsCross.fuloongminipc
pkgsCross.ghcjs
pkgsCross.gnu32
pkgsCross.gnu64
pkgsCross.i686-embedded
pkgsCross.iphone32
pkgsCross.iphone32-simulator
pkgsCross.iphone64
pkgsCross.iphone64-simulator
pkgsCross.mingw32
pkgsCross.mingw64
pkgsCross.mmix
pkgsCross.msp430
pkgsCross.remarkable2
pkgsCross.riscv32
pkgsCross.riscv32-embedded
pkgsCross.riscv64
pkgsCross.riscv64-embedded
pkgsCross.scaleway-c1
pkgsCross.sheevaplug
pkgsCross.vc4
pkgsCross.wasi32
pkgsCross.x86_64-embedded
pkgsCross.x86_64-netbsd
pkgsCross.x86_64-netbsd-llvm
pkgsCross.x86_64-unknown-redox
```

These attribute names for cross compilation packages have been chosen somewhat freely over the course of time. They usually do not match the corresponding platform config string.

You can retrieve the platform string from

```
pkgsCross.<platform>.stdenv.hostPlatform.config:
```

```
nix-repl> pkgsCross.aarch64-multiplatform.stdenv.hostPlatform.config
"aarch64-unknown-linux-gnu"
```

If the host platform you seek hasn't been defined yet, please [contribute it upstream](#).

Specifying the host platform

The mechanism for setting up cross compilation works as follows:

1. Take the build platform configuration and apply it to the current package set, called `pkgs` by convention.

The build platform is implied in `pkgs = import <nixpkgs> {}` to be the current system. This produces a build environment `pkgs.stdenv` with all the dependencies present to compile on the build platform.

2. Apply the appropriate host platform configuration to all the packages in `pkgsCross`.

Taking `pkgs.pkgsCross.<host>.hello` will produce the package `hello` compiled on the build platform to run on the `<host>` platform.

There are multiple equivalent ways to access packages targeted to the host platform.

1. Explicitly pick the host platform package from within the build platform environment:

[Skip to main content](#)

```
1 let
2   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23";
3   pkgs = import nixpkgs {};
4 in
5   pkgs.pkgsCross.aarch64-multiplatform.hello
```

2. Pass the host platform to `crossSystem` when importing `nixpkgs`. This configures `nixpkgs` such that all its packages are build for the host platform:

```
1 let
2   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23";
3   pkgs = import nixpkgs { crossSystem = { config = "aarch64-unknown-linux-gnu"; }; };
4 in
5   pkgs.hello
```

Equivalently, you can pass the host platform as an argument to `nix-build`:

```
$ nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 \
--arg crossSystem '{ config = "aarch64-unknown-linux-gnu"; }' \
-A hello
```

Cross compiling for the first time

To cross compile a package like `hello`, pick the platform attribute — `aarch64-multiplatform` in our case — and run:

```
$ nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 \
-A pkgsCross.aarch64-multiplatform.hello
...
/nix/store/1dx87l5rav8679lqigf9xxkb7wvh2m4k-hello-aarch64-unknown-linux-gnu-2.12.1
```

Note

The hash of the package in the store path changes with the updates to the channel.

Search for a [package](#) attribute name to find the one you're interested in building.

Real-world cross compiling of a Hello World

[Skip to main content](#)

example

To show off the power of cross compilation in Nix, let's build our own Hello World program by cross compiling it as static executables to `armv6l-unknown-linux-gnueabi` and `x86_64-w64-mingw32` (Windows) platforms and run the resulting executable with an emulator.

Given we have a `cross-compile.nix`:

```
1 let
2   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
3   pkgs = import nixpkgs {};
4
5   # Create a C program that prints Hello World
6   helloWorld = pkgs.writeText "hello.c" ''
7     #include <stdio.h>
8
9     int main (void)
10    {
11      printf ("Hello, world!\n");
12      return 0;
13    }
14  '';
15
16  # A function that takes host platform packages
17  crossCompileFor = hostPkgs:
18    # Run a simple command with the compiler available
19    hostPkgs.runCommandCC "hello-world-cross-test" {} ''
20      # Wine requires home directory
21      HOME=$PWD
22
23      # Compile our example using the compiler specific to our host platform
24      $CC ${helloWorld} -o hello
25
26      # Run the compiled program using user mode emulation (Qemu/Wine)
27      # buildPackages is passed so that emulation is built for the build platform
28      ${hostPkgs.stdenv.hostPlatform.emulator hostPkgs.buildPackages} hello > $out
29
30      # print to stdout
31      cat $out
32    '';
33  in {
34    # Statically compile our example using the two platform hosts
35    rpi = crossCompileFor pkgs.pkgsCross.raspberryPi;
36    windows = crossCompileFor pkgs.pkgsCross.mingwW64;
37  }
```

If we build this example and print both resulting derivations, we should see "Hello, world!" for each:

[Skip to main content](#)

```
$ cat $(nix-build cross-compile.nix)
Hello, world!
Hello, world!
```

Developer environment with a cross compiler

In the [tutorial for declarative reproducible environments](#), we looked at how Nix helps us provide tooling and system libraries for our project.

It's also possible to provide an environment with a compiler configured for **cross-compilation to static binaries using musl**.

Given we have a `shell.nix`:

```
1 let
2   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
3   pkgs = (import nixpkgs {}).pkgsCross.aarch64-multiplatform;
4 in
5
6 # callPackage is needed due to https://github.com/NixOS/nixpkgs/pull/126844
7 pkgs.pkgsStatic.callPackage ({ mkShell, zlib, pkg-config, file }: mkShell {
8   # these tools run on the build platform, but are configured to target the host
9   nativeBuildInputs = [ pkg-config file ];
10  # libraries needed for the host platform
11  buildInputs = [ zlib ];
12 }) {}
```

And `hello.c`:

```
#include <stdio.h>

int main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

We can cross compile it:

```
$ nix-shell --run '$CC hello.c -o hello' shell.nix
```

[Skip to main content](#)

```
$ nix-shell --run 'file hello' shell.nix
hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically linked,
```

Next steps

- The [official binary cache](#) has a limited number of binaries for packages that are cross compiled, so to save time recompiling, configure [your own binary cache and CI with GitHub Actions](#).
- While many compilers in Nixpkgs support cross compilation, not all of them do. Additionally, supporting cross compilation is not trivial work and due to many possible combinations of what would need to be tested, some packages might not build.

[A detailed explanation how of cross compilation is implemented in Nix](#) can help with fixing those issues.

- The Nix community has a [dedicated Matrix room](#) for help with cross compiling.

[1] Terminology for cross compilation platforms differs between build systems. We have chosen to follow [autoconf terminology](#).