GCC Cross-Compiler

From OSDev Wiki

This page or section refers to its readers or editors using *I*, *my*, *we* or *us*. It should be edited (https://osdev.org/index.php?title=GCC_Cross-Compiler&action=edit) to be in an encyclopedic tone.

Difficulty level

🟩⬜⬜⬜
Beginner

This tutorial focuses on creating a GCC cross-compiler for your own operating system. This compiler that we build here will have a generic target (i686-elf) that allows you to leave the current operating system behind, meaning that no headers or libraries of the host operating system will be used. Without using a cross-compiler for operating system development, a lot of unexpected things can happen because the compiler assumes that the code is running on the host operating system.

Contents

Introduction

Generally speaking, a cross-compiler is a compiler that runs on platform A (the **host**), but generates executables for platform B (the **target**). These two platforms may (but do not need to) differ in CPU, operating system, and/or executable format. In our case, the host platform is your current operating system and the target platform is the operating system you are about to make. It is important to realize that these two platforms are not the same; the operating system you are developing is always going to be different from the operating system you currently use.

Why cross-compilers are necessary

Main article: Why do I need a Cross Compiler?

The compiler must know the correct target platform (CPU, operating system). The compiler that comes with the host system does not know by default that it is compiling something else entirely, unless a lot of problematic options are passed to it, which will create a lot of problems in the future. The solution is to build a cross-compiler.

Which compiler version to choose

Main article: Building GCC

The newest GCC is recommended as it is the latest and greatest release. For instance, using GCC 4.6.3 to build a GCC 4.8.0 cross-compiler would create troubles. Here is how to build the newest GCC as your system compiler.

You can also use older releases as they are usually reasonably good. If your local system compiler isn't too terribly old (at least GCC 4.6.0), you may wish to save yourself the trouble and just pick the latest minor release (such as 4.6.3 if your system compiler is 4.6.1) for your cross-compiler.

This command prints the current compiler version:

```
gcc --version
```

You may be able to use an older major GCC release to build a cross-compiler of a newer major GCC release. For instance, GCC 4.7.3 may be able to build a GCC 4.8.0 cross-compiler. However, if you want to use the latest and greatest GCC version for your cross-compiler, we recommend that you bootstrap the newest GCC as your system compiler first. Individuals using OS X 10.7 or earlier might want to invest in either building a system GCC (that outputs native Mach-O), or upgrading the local LLVM/Clang installation. Users with 10.8 and above should install the Command Line Tools from Apple's developer website and use Clang to cross-compile GCC.

Which binutils version to choose

Main article: Cross-Compiler Successful Builds

The latest and greatest Binutils release is recommended. Note, however, that not all combinations of GCC and Binutils work. If you run into trouble, use a Binutils that was released at roughly the same time as your desired compiler version. You probably need at least Binutils 2.22, or preferably the latest 2.23.2 release. It doesn't matter what Binutils version you have installed on your current operating system. This command prints the binutils version:

```
ld --version
```

Deciding on the target platform

Main article: Target Triplet

You should already know this. If you are following the Bare Bones tutorial, you wish to build a cross-compiler for `i686-elf`.

Note on arm-none-eabi-gcc

There is the prebuilt package gcc-arm-none-eabi on apt-get for Debian/Ubuntu, but you shouldn't use it because it neither contains a libgcc.a nor freestanding C header files like stdint.h.
Instead you should build it yourself with `arm-none-eabi` being the $TARGET.

Preparing for the build

Difficulty level

The GNU Compiler Collection is an advanced piece of software with dependencies. You need the following in order to build GCC:

- A Unix-like environment (Windows users can use the Windows Subsystem for Linux or Cygwin)
- Enough memory and hard disk space (it depends, 256 MiB will not be enough).
- GCC (existing release you wish to replace), or another system C compiler
- G++ (if building a version of GCC >= 4.8.0), or another system C++ compiler
- Make
- Bison
- Flex
- GMP
- MPFR
- MPC
- Texinfo
- ISL (optional)
- CLooG (optional)

Beginner

Installing Dependencies

↓ Dependency / OS →	Source Code	Debian (Ubuntu, Mint, WSL, ...)	Gentoo	Fedora	Cygwin	OpenBSD	Arch
How to install	Normally	`sudo apt install` ***foo***	`sudo emerge --ask` ***foo***	`sudo dnf install` ***foo***	Cygwin GUI setup	`doas pkg_add` ***foo***	`pacman -Syu` ***foo***
Compiler	N/A	`build-essential`	`sys-devel/gcc`	`gcc gcc-c++`	`mingw64-x86_64-gcc-g++` / `mingw64-i686-gcc-g++`	Preinstalled	`base-devel`
Make	N/A	`build-essential`	`sys-devel/make`	`make`	`make`	Preinstalled	`base-devel`
Bison (https://www.gnu.org/software/bison/)	[1] (https://ftp.gnu.org/gnu/bison/)	`bison`	`sys-devel/bison`	`bison`	`bison`	?	`base-devel`
Flex (https://github.com/westes/flex)	[2] (https://github.com/westes/flex/releases)	`flex`	`sys-devel/flex`	`flex`	`flex`	?	`base-devel`
GMP (https://gmplib.org/)	[3] (https://ftp.gnu.org/gnu/gmp/)	`libgmp3-dev`	`dev-libs/gmp`	`gmp-devel`	`libgmp-devel`	`gmp`	`gmp`
MPC	[4] (https://ftp.gnu.org/gnu/mpc/)	`libmpc-dev`	`dev-libs/mpc`	`libmpc-devel`	`libmpc-devel`	`libmpc`	`libmpc`
MPFR (https://www.mpfr.org/)	[5] (https://ftp.gnu.org/gnu/mpfr/)	`libmpfr-dev`	`dev-libs/mpfr`	`mpfr-devel`	`libmpfr-devel`	`mpfr`	`mpfr`
Texinfo (https://www.gnu.org/software/texinfo/)	[6] (https://ftp.gnu.org/gnu/texinfo/)	`texinfo`	`sys-apps/texinfo`	`texinfo`	`texinfo`	`texinfo`	`base-devel`
CLooG (https://www.cloog.org/) (Optional)	CLooG (https://www.cloog.org/)	`libcloog-isl-dev`	`dev-libs/cloog`	`cloog-devel`	`libcloog-isl-devel` (Deprecated, no longer exists!)	N/A	N/A
ISL (http://isl.gforge.inria.fr/) (Optional)	[7] (http://isl.gforge.inria.fr/)	`libisl-dev`	`dev-libs/isl`	`isl-devel`	`libisl-devel`	N/A	N/A

You need to have Texinfo installed to build Binutils. You need to have GMP, MPC, and MPFR installed to build GCC. GCC optionally can make use of the CLooG and ISL libraries.

For instance, you can install `libgmp3-dev` on Debian by running the shell command: `sudo apt install libgmp3-dev`

Note: Version 5.x (or later) of Texinfo is known to be incompatible with the current Binutils 2.23.2 release (and older). You can check your current version using `makeinfo --version`. If your version is too new and you encounter problems during the build, you will need to either use Binutils 2.24 release (or newer) or install an older version of Texinfo - perhaps through building from source - and add it to your `PATH` prior and during the Binutils build.

Note: Version 0.13 (or later) of ISL is incompatible with the current CLooG 0.18.1 release (and older). Use version 0.12.2 of ISL or the build **will** fail.

Note: If you are using **Cygwin**, it is recommended to install the **libintl-devel** package (I couldn't build a cross compiler without this package)

Downloading the Source Code

Download the needed source code into a suitable directory such as `$HOME/src`:

- You can download the desired Binutils release by visiting the Binutils website (https://gnu.org/software/binutils/) or directly accessing the GNU main mirror (https://ftp.gnu.org/gnu/binutils/) .
- You can download the desired GCC release by visiting the GCC website (https://gnu.org/software/gcc/) or directly accessing the GNU main mirror (https://ftp.gnu.org/gnu/gcc/) .

Note: The versioning scheme used is that each fullstop separates a full number, i.e. Binutils 2.20.0 is newer than 2.9.0. This may be confusing, if you have not encountered this (quite common) versioning scheme yet, when looking at an alphanumerically sorted list of tarballs: The file at the bottom of the list is not the latest version! An easy way of getting the latest version is to sort by the last modified date and scrolling to the bottom.

Linux Users building a System Compiler

Your distribution may ship its own patched GCC and Binutils that is customized to work on your particular Linux distribution. You should be able to build a working cross-compiler using the above source code, but you might not be able to build a new system compiler for your current Linux distribution. In that case, try a newer GCC release or get the patched source code.

Gentoo Users

Gentoo offers crossdev (https://wiki.gentoo.org/wiki/Crossdev) to set up a cross-development toolchain:

```
emerge -av crossdev
crossdev --help
PORTDIR_OVERLAY="/usr/local/crossdev" crossdev --stage1 --binutils <binutils-version> --gcc <gcc-version> --target <target>
```

This will install a GCC cross-compiler into a "slot", i.e. alongside already-existing compiler versions. You can install several cross-compilers that way, simply by changing target designations. An unfortunate downside is that it will also pull in gentoo patches and pass additional configure options that differ from the official **GCC Cross-Compiler** setup, and they might behave differently.

After the compilation finishes successfully, your cross-compiler is callable via <target>-gcc. You can also use gcc-config to toggle between compiler versions should you need to do so. Don't replace your system compiler with a cross-compiler. The package manager will also suggest updates as soon as they become available.

You can uninstall the cross-compiler by calling *crossdev --clean <target>*. Read the cross-development (https://wiki.gentoo.org/wiki/Cross_build_environment) document for additional information.

Note that the version numbers to binutils and gcc are *Gentoo package versions*, i.e. there might be a suffix to the "official" (GNU) version that addresses additional patchsets supplied by the Gentoo maintainers. (For example, *--binutils 2.24-r3 --gcc 4.8.3* is the latest stable package pair at the time of this writing.) You can omit the version numbers to use the latest package available.

Portage uses overlays to store packages that are not part of the original package management. Crossdev needs one overlay where it can store its binutils and gcc packages before building them. You can configure one properly, or you can use PORTDIR_OVERLAY to point at where it should keep its package manager files. Using PORTDIR_OVERLAY is not a good idea with existing overlays, but by then you should know how you have personally set them up earlier anyway and how to do it properly. See [8] (https://wiki.gentoo.org

/wiki/Custom_repository#Crossdev) .

macOS Users

macOS users need a replacement libiconv because the system libiconv is seriously out of date. macOS users can download the latest libiconv release by visiting the libiconv website (https://gnu.org/software/libiconv/) or directly accessing the GNU main FTP mirror (https://ftp.gnu.org/gnu/libiconv/) . Otherwise you may get unresolved symbol errors related to libiconv when compiling GCC 4.3 or higher on OS X 10.4 and 10.5.

Install a new version (compile it yourself or use MacPorts) and add `--with-libiconv-prefix=/opt/local` (or `/usr/local` if you compiled it yourself) to GCC's `./configure` line. Alternatively you may place the libiconv source as gcc-x.y.z/libiconv and it will be compiled as part of the GCC compilation process. (This trick also works for MPFR, GMP, and MPC).

The makefiles of Binutils and GCC use the `$(CC)` variable to invoke the compiler. On OS X, this resolves to `gcc` by default, which is actually `clang`. Prior to OS X 10.8, the Clang that came with Xcode's Command Line Tools package was not able to build a working GCC. Users running OS X 10.7 or below may need to find and install GCC, either from Homebrew (https://brew.sh) , or from somewhere on Apple's website. You can try with the old GCC that comes preinstalled on some macOS versions.

```
# This is only necessary for OS X users running 10.7 or below.
export CC=/usr/bin/gcc-4.2
export CXX=/usr/bin/g++-4.2
export CPP=/usr/bin/cpp-4.2
export LD=/usr/bin/gcc-4.2
```

You will want to unset these exports once you compiled and installed the cross compiler.

Note for Lion users: If you're on Lion (or above) chances are that you don't have the "real" GCC since Apple removed it from the Xcode package, but you can still install it. You can do it via Homebrew or by compiling from source, both are perfectly described on a StackExchange answer (https://apple.stackexchange.com/a/38247) .

Note for Maverick users: You can build binutils-2.24 and gcc-4.8.3 (possible other version) with Xcode 5.1.1. Note that building GCC with LLVM is not officially supported and may cause interesting bugs, if you are willing to take this risk and save time building host-gcc just to compile a cross-gcc, follow this. Install GMP, MPFR, Mpc with MacPorts (https://www.macports.org/) .

```
sudo port install gmp mpfr libmpc
```

```
../binutils-2.24/configure --prefix=$PREFIX \
--target=$TARGET \
--enable-interwork --enable-multilib \
--disable-nls --disable-werror
```

```
../gcc-4.8.3/configure --prefix=$PREFIX \
--target=$TARGET \
--disable-nls \
--enable-languages=c,c++ --without-headers \
--enable-interwork --enable-multilib \
--with-gmp=/usr --with-mpc=/opt/local --with-mpfr=/opt/local
```

Note: There is an issue with port's GMP, so the version from OS X from /usr is used instead. **Note2:** If you still have some errors, try making a case-sensitive APFS disk image using disk utility app and build from there

Windows Users

Windows users need to set up a Unix-like enviroment such as MinGW or Cygwin. It may well be worth looking into systems such as Linux and see if they fit your needs, as you commonly use a lot of Unix-like tools in operating systems development and this is much easier from a Unix-like operating system. **If you have just installed the basic Cygwin package, you have to run the setup.exe again and install the following packages:** GCC, G++, Make, Flex, Bison, Diffutils, libintl-devel, libgmp-devel, libmpfr-devel, libmpc-devel, Texinfo

MinGW + MSYS is an option, and as it addresses the native Windows API instead of a POSIX emulation layer, results in a slightly faster toolchain. Some software packages will not build properly under MSYS as they were not designed for use with Windows. As far as this tutorial is concerned, everything that applies to Cygwin also applies to MSYS unless otherwise specified. Make sure you install the C and C++ compilers, and the MSYS Basic System.

The "Windows Subsystem for Linux (Beta)", released with the Windows 10 Anniversary update is also an option for using a cross compiler. (Tested 08/08/2016 with GCC 6.1.0 and Binutils 2.27) This cross-compiler works reasonably fast, although being in beta state, it may not be ideal permanent development platform.

Cygwin note: Cygwin includes your Windows `%PATH%` in its bash `$PATH`. If you were using DJGPP before, this could result in confusion as e.g. calling `GCC` on the Cygwin bash command line would still call the DJGPP compiler. After uninstalling DJGPP, you should delete the DJGPP environment variable and clear the `C:\djgpp` entry (or wherever you installed it) from your `%PATH%`. Likewise, it might be a bad idea to mix build environments in your system PATH variable.

MinGW note: Some MinGW-specific information on building a cross-toolchain can be found on the hosted cross-compiler how-to page (http://www.mingw.org/wiki/HostedCrossCompilerHOWTO) on the MinGW homepage.

Windows Subsystem for Linux (Beta) Note: You cannot have your cross compiler in the /mnt/c/ (or /mnt/"x") areas, as trying to compile your cross-compiler there will generate errors, whereas building to $HOME/opt/cross works perfectly. This is fixed with Windows Update KB3176929

OpenBSD Users

OpenBSD users might need to install "gcc" package from ports because base system's GCC is very outdated. If you want to build GCC, try to use the ports' version instead of the latest version available and apply all patches from ports to your build. Also, if the build fails during compiling lto-plugin, a temporary solution is to disable LTO altogether during configure stage of building GCC by adding `--disable-lto`

The Build

We build a toolset running on your host that can turn source code into object files for your target system.

You need to decide where to install your new compiler. It is dangerous and a very bad idea to install it into system directories. You also need to decide whether the new compiler should be installed globally or just for you. If you want to install it just for you (recommended), installing into `$HOME/opt/cross` is normally a good idea. If you want to install it globally, installing it into `/usr/local/cross` is normally a good idea.

Please note that we build everything out of the source directory tree, as is considered good practice. Some packages only support building outside, some only inside and some both (but may not offer extensive checking with make). Building GCC inside the source directory tree fails miserably, at least for older versions.

Preparation

```
export PREFIX="$HOME/opt/cross"
export TARGET=i686-elf
export PATH="$PREFIX/bin:$PATH"
```

We add the installation prefix to the `PATH` of the current shell session. This ensures that the compiler build is able to detect our new binutils once we have built them.

The prefix will configure the build process so that all the files of your cross-compiler environment end up in $HOME/opt/cross. You can change that prefix to whatever you like (e.g., /opt/cross or $HOME/cross would be options). If you have administrator access and wish to make the cross-compiler toolchain available to all users, you can install it into the /usr/local prefix - or perhaps a /usr/local/cross prefix if you are willing to change the system configuration such that this directory is in the search paths for all users. Technically, you could even install directly to /usr, so that your cross-compiler would reside alongside your system compiler, but that is not recommended for several reasons (like risking to overwrite your system compiler if you get TARGET wrong, or getting into conflict with your system's package management).

Binutils

```
cd $HOME/src

mkdir build-binutils
cd build-binutils
../binutils-x.y.z/configure --target=$TARGET --prefix="$PREFIX" --with-sysroot --disable-nls
make
make install
```

This compiles the binutils (assembler, disassembler, and various other useful stuff), runnable on your system but handling code in the format specified by $TARGET.

--disable-nls tells binutils not to include native language support. This is basically optional, but reduces dependencies and compile time. It will also result in English-language diagnostics, which the people on the Forum (http://forum.osdev.org/) understand when you ask your questions. ;-)

--with-sysroot tells binutils to enable sysroot support in the cross-compiler by pointing it to a default empty directory. By default, the linker refuses to use sysroots for no good technical reason, while gcc is able to handle both cases at runtime. This will be useful later on.

GDB

It may be worth noting that if you wish to use ```GDB```, and you are running on a different computer architecture than you OS (most common case is developing for ARM on x86_64 or x86_64 on ARM), you need to cross-compile GDB separately. While technically a part of Binutils, resides in a separate repository.

The protocol for building GDB to target a different architecture is very similar to that of regular Binutils:

```
../gdb.x.y.z/configure --target=$TARGET --prefix="$PREFIX" --disable-werror
make all-gdb
make install-gdb
```

The ```--disable-nls``` and ```--with-sysroot`` options don't seem to have any effect.

GCC

See also the offical instructions for configuring gcc (http://gcc.gnu.org/install/configure.html) .

Now, you can build GCC.

```
cd $HOME/src

# The $PREFIX/bin dir _must_ be in the PATH. We did that above.
which -- $TARGET-as || echo $TARGET-as is not in the PATH

mkdir build-gcc
cd build-gcc
../gcc-x.y.z/configure --target=$TARGET --prefix="$PREFIX" --disable-nls --enable-languages=c
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

We build libgcc, a low-level support library that the compiler expects available at compile time. Linking against libgcc provides integer, floating point, decimal, stack unwinding (useful for exception handling) and other support functions. Note how we are *not* simply running `make && make install` as that would build way too much, not all components of gcc are ready to target your unfinished operating system.

--disable-nls is the same as for binutils above.

--without-headers tells GCC not to rely on any C library (standard or runtime) being present for the target.

--enable-languages tells GCC not to compile all the other language frontends it supports, but only C (and optionally C++).

It will take a while to build your cross-compiler. On a multi-core machine, speed up the build by parallelizing it, e.g. `make -j 8 all-gcc`, if 8 is the number of jobs to run simultaneously.

If you are building a cross compiler for x86-64, you may want to consider building Libgcc without the "red zone": Libgcc_without_red_zone

Using the new Compiler

Now you have a "naked" cross-compiler. It does not have access to a C library or C runtime yet, so you cannot use most of the standard includes or create runnable binaries. But it is quite sufficient to compile the kernel you will be making shortly. Your toolset resides in $HOME/opt/cross (or what you set `$PREFIX` to). For example, you have a GCC executable installed as `$HOME/opt/cross/bin/$TARGET-gcc`, which creates programs for your TARGET.

You can now run your new compiler by invoking something like:

```
$HOME/opt/cross/bin/$TARGET-gcc --version
```

Note how this compiler is *not* able to compile normal C programs. The cross-compiler will spit errors whenever you want to #include any of the standard headers (except for a select few that actually are platform-independent, and generated by the compiler itself). This is quite correct - you don't have a standard library for the target system yet!

The C standard defines two different kinds of executing environments - "freestanding" and "hosted". While the definition might be rather fuzzy for the average application programmer, it is pretty clear-cut when you're doing OS development: A kernel is "freestanding", everything you do in user space is "hosted". A "freestanding" environment needs to provide only a subset of the C library: `float.h`, `iso646.h`, `limits.h`, `stdalign.h`, `stdarg.h`, `stdbool.h`, `stddef.h`, `stdint.h` and `stdnoreturn.h` (as of C11). All of these consist of typedef s and #define s "only", so you can implement them without a single .c file in sight.

Note that to have these compiler-provided includes work properly, you need to build your kernel with the `-ffreestanding` flag. Otherwise, they might attempt including your standard library's copy of the headers, which isn't gonna work if you don't have a standard library.

To use your new compiler simply by invoking `$TARGET-gcc`, add `$HOME/opt/cross/bin` to your `$PATH` by typing:

```
export PATH="$HOME/opt/cross/bin:$PATH"
```

This command will add your new compiler to your PATH for this shell session. If you wish to use it permanently, add the PATH command to your `~/.profile` configuration shell script or similar. Consult your shell documentation for more information.

You can now move on to complete the Bare Bones tutorial variant that lead you here and complete it using your new cross-compiler. If you built a new GCC version as your system compiler and used it to build the cross-compiler, you can now safely uninstall it unless you wish to continue using it.

Troubleshooting

In general, **verify** that you read the instructions carefully and typed the commands precisely. Don't skip instructions. You will have to set your PATH variable again if you use a new shell instance, if you don't make it permanent by adding it to your shell profile. If a compilation seems to have gotten really messed up, type `make distclean`, and then start the make process over again. Ensure your un-archiever doesn't change newline characters.

ld: cannot find -lgcc

You specified that you want to link the GCC low-level runtime library into your executable through the `-lgcc`' switch, but forgot to build and properly install the library.

If you got no warnings or errors while installing libgcc and you still have problems you can copy the library in your project and link it with `-L. -lgcc`
The libgcc is at $PREFIX/lib/gcc/$TARGET/<gcc-version>/libgcc.a .

Binutils 2.9

What's alphabetically on the top or bottom is not necessarily the latest version. After 2.9 comes 2.10, 2.11, 2.12 and then there are more releases that are all newer and progressively more likely to build or support your choice of GCC version.

Building GCC: the directory that should contain system headers does not exist

You might encounter this error when building `mingw32` targets, for example `x86_64-w64-mingw32`. The offending directory that can't be found is `$SYSROOT/mingw/include`. If you look in your sysroot, you will, of course, realise that no such folder exists.

The solution is simply to create the empty folders:

```
mkdir -p $SYSROOT/mingw/include
mkdir -p $SYSROOT/mingw/lib
```

This will allow the build to proceed. The reason this happens is that the `mingw32` (and mingw itself) configures `INCLUDE_PATH` and `LIBRARY_PATH` to be, as can be guessed, `/mingw/include` and `/mingw/lib`, instead of the defaults `/usr/include` and `/usr/lib`. Why the build fails even though nothing is required in those folders, and why it doesn't just make them, is beyond me.

GCC libsanitizer failing to build

Sometimes GCC can't build libsanitizer, if that happens append `--disable-libsanitizer` to the configure command. This only applies for building a hosted compiler.

More advanced

Using this simple cross compiler will be sufficient for quite some time, but at some point you will want the compiler to automatically include your own system headers and libraries. Building an OS-specific toolchain for your own operating system is the way to go from here.

See Also

Articles

- Cross-Compiler Successful Builds - combinations of GCC and Binutils which have been shown to work with this tutorial by OSDev.org members.
- Target Triplet - on target triplets and their use
- OS Specific Toolchain - going a step further and adding your own target.
- LLVM Cross-Compiler - some compilers make things much easier.
- Canadian Cross - making things yet more complicated.

External Links

- http://kegel.com/crosstool has a popular example of a script that automatically downloads, patches, and builds binutils, gcc, and glibc for known platforms.
- http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html - Summary of the support functions you get when you link with libgcc.
- http://forums.gentoo.org/viewtopic.php?t=66125 - Compiling Windows applications under Linux
- http://www.libsdl.org/extras/win32/cross/README.txt - dito
- https://github.com/travisg/toolchains - Another script for building simple cross compilers
- https://www.youtube.com/watch?v=aESwsmnA7Ec - A walkthrough of how to build a cross-compiler using Cygwin on Windows.
- https://github.com/Cheapskate01/Cross-Compiler-Build-Script - A dead-simple script that Binutils and Gcc

for you.

Prebuilt Toolchains

These were built by people in the OSdev community for their own building needs and shared at will, without guaranteeing any support or that it will even work on your setup. YMMV.

Latests versions for Linux (many arch)

- kernel.org various hosts/targets (https://www.kernel.org/pub/tools/crosstool/)

For Linux i686 host

- aarch64-elf 9.3.0 target (https://drive.google.com/open?id=1zcFAmxi7mtOwhMaKE36IsjLRB0uEv17p)
- arm-eabi 9.3.0 target (https://drive.google.com/open?id=1uEFrOJPxy13vxWCy-5IbxCJAs5TRdtTE)
- i386-elf 9.3.0 target (https://drive.google.com/open?id=13Kg6Xd8acUnwUoZQBTOjwAQQeGOeYzVz)
- i486-elf 9.3.0 target (https://drive.google.com/open?id=1F5RsfIEfcpRYAqu5UuGTKgkMa3eBsXnP)
- i586-elf 9.3.0 target (https://drive.google.com/open?id=1PdEFqMEJf_Vuf0drO1m8bjlrsudAmI5k)
- i686-elf 9.3.0 target (https://drive.google.com/open?id=1g9jzEIn8CB6ZiVrc0uxbZprepcX4gYex)
- mips-elf 9.3.0 target (https://drive.google.com/open?id=1xIeNJwD0Do-REFxzCLOkMIVYI3HGJUTX)
- mips64-elf 9.3.0 target (https://drive.google.com/open?id=10UuOf9LW4y9WEJcTWrMZW0GAlOWh9KG7)
- m64k-elf 9.3.0 target (https://drive.google.com/open?id=1oW6UWr-OY22EgtyuXovORUtHeVLkL-Tw)
- powerpc-elf 9.3.0 target (https://drive.google.com/open?id=1H3Cbq4D_kjROB-7Otiisa-mac_9FYUM-)
- sh-elf 9.3.0 target (https://drive.google.com/open?id=1kFGNHWhcBD9cf8X2y-BW49Su04lMa1ev)
- sparc-elf 9.3.0 target (https://drive.google.com/open?id=1XzQDKTK35EX8b380bPRbuhX1GPKrtI77)
- x86_64-elf 9.3.0 target (https://drive.google.com/open?id=1pU8bS0McTyRUTHb1b_c8ZVa2Ka_XeeEp)
- xtensa-elf 9.3.0 target (https://drive.google.com/open?id=1cqk9RzY3QXQuaS-YdIWtCECKL81pcsv-)

For Linux x86_64 host

- i386-elf & i686-elf 7.1.0 target uploaded by TheAlmostGenius (https://drive.google.com/file/d/0Bw6lG3Ej2746STJaM2dNbC05elE/view?usp=sharing)

- i386-elf 7.5.0 target (https://newos.org/toolchains/i386-elf-7.5.0-Linux-x86_64.tar.xz)
- x86_64-elf 7.5.0 target (https://newos.org/toolchains/x86_64-elf-7.5.0-Linux-x86_64.tar.xz)
- aarch64-elf 7.5.0 target (https://newos.org/toolchains/aarch64-elf-7.5.0-Linux-x86_64.tar.xz)
- arm-eabi 7.5.0 target (https://newos.org/toolchains/arm-eabi-7.5.0-Linux-x86_64.tar.xz)
- m68k-elf 7.5.0 target (https://newos.org/toolchains/m68k-elf-7.5.0-Linux-x86_64.tar.xz)
- microblaze-elf 7.5.0 target (https://newos.org/toolchains/microblaze-elf-7.5.0-Linux-x86_64.tar.xz)
- mips-elf 7.5.0 target (https://newos.org/toolchains/mips-elf-7.5.0-Linux-x86_64.tar.xz)
- nios2-elf 7.5.0 target (https://newos.org/toolchains/nios2-elf-7.5.0-Linux-x86_64.tar.xz)
- powerpc-elf 7.5.0 target (https://newos.org/toolchains/powerpc-elf-7.5.0-Linux-x86_64.tar.xz)
- riscv32-elf 7.5.0 target (https://newos.org/toolchains/riscv32-elf-7.5.0-Linux-x86_64.tar.xz)
- riscv64-elf 7.5.0 target (https://newos.org/toolchains/riscv64-elf-7.5.0-Linux-x86_64.tar.xz)
- sh-elf 7.5.0 target (https://newos.org/toolchains/sh-elf-7.5.0-Linux-x86_64.tar.xz)
- sparc-elf 7.5.0 target (https://newos.org/toolchains/sparc-elf-7.5.0-Linux-x86_64.tar.xz)

The packages from phillid.tk below have been shrunk to about 10 MiB for each pair of packages (GCC & Binutils). Please note that this has been achieved by enabling only the C front-end for GCC. If you're going to write your OS in any language but C or Assembly, these packages aren't for you. These are actually Pacman packages, but untarring them to / and rm-ing /.MTREE and other clutter dotfiles contained in the package will work the same.

For Windows host

- i686-elf 4.8.2 target (https://drive.google.com/file/d/0B85K_c7mx3QjUnZuaFRPWlBIcXM/edit?usp=sharing)
- x86_64-elf 5.1.0 target (https://mega.co.nz/#F!bBxA3SKJ!TDL4i1NjaZKd4YMo9p2U7g)
- i686-/x86_64-elf 13.2.0 target + GDB (Windows/Linux/WSL) (https://github.com/lordmilko/i686-elf-tools)

For Windows Subsystem for Linux (Beta) host

- i686-elf 6.1.0 target (http://www.bin-os.com/i686-elf-6.1.0.tar.gz) (extracts to a directory called "cross", don't forget to install 'make' - I would recommend "apt-get install build-essential" to also add additional useful tools)

For Windows Subsytem for Linux host

- i686-elf 13.1.0 target (https://drive.google.com/file/d/1zn37YmyVrtsPOfe88jZRZ_-_3_jzpXBP/view?usp=sharing) (**Important: extracts to a directory called "cross-tk"**)

For macOS host

x86_64-elf binutils (https://formulae.brew.sh/formula/x86_64-elf-binutils) and gcc (https://formulae.brew.sh/formula/x86_64-elf-gcc) (canonical target name x86_64-pc-elf) can be installed from homebrew (https://brew.sh) :

```
$ brew install x86_64-elf-gcc
```

i686-elf toolchain is also available (https://formulae.brew.sh/formula/i686-elf-gcc) in homebrew.

ARM prebuilt toolchains for multiple host platforms

ARM provides it's own prebuilt toolchain based upon GNU utilities for development targeting ARM systems.

- GNU ARM Embedded Toolchain (https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads)

Docker image

- i686-elf 8.1.0 target (https://hub.docker.com/r/joshwyant/gcc-cross/)

Retrieved from "https://wiki.osdev.org/index.php?title=GCC_Cross-Compiler&oldid=28436"
Categories: Articles Written in First Person | Level 1 Tutorials | Compilers | Tutorials

- This page was last modified on 2 November 2023, at 17:32.
- This page has been accessed 1,146,022 times.