Build a GCC-based cross compiler for Linux

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

There are times when the platform you're developing on and the computer you're developing for don't match. For example, you might want to build a PowerPC/Linux application from your x86/Linux laptop. Using the gcc, gas, and ld tools from the GNU toolkits, you can specify and build a cross-compiler that will enable you to build for other targets on your machine. With a bit more work, you can even set up an environment that will build an application for a variety of different targets. In this tutorial, I describe the process required to build a cross-compiler on your system. I also discuss building a complete environment for a range of targets, show you how to integrate with the distcc and ccache tools, and describe the methods required to keep up-to-date with the latest revisions and updates on your new development platform.

To build a cross-compiler, you need a basic knowledge of the build process of a typical UNIX open source project, some basic shell skills, and a lot of patience.

Prerequisites

To build a cross-compiler, you need a working C compiler (gcc is generally a good idea). A C compiler is supplied with most Linux /UNIX-based operating systems. You also need the source code for the various tools used to build the cross-compiler. You can download GNU tools from *GNU* (http://www.gnu.org) . (Use a local mirror where possible.)

In addition to these tools, you need a copy of the header files for your target platform. For a Linux target, use the generic Linux kernel headers available from *Kernel.org* (http://www.kernel.org) .

About the author

Martin C. Brown is a former IT Director with experience in cross-platform integration. A keen developer, he has produced dynamic sites for blue-chip customers, including HP and Oracle, and is the Technical Director of Foodware.net. Now a freelance writer and consultant, MC (as he is better known) works closely with Microsoft as an SME, is the LAMP Technologies Editor for *LinuxWorld* magazine, is a core member of the AnswerSquad.com

team, and has written several books on topics as diverse as Microsoft certification, iMacs, and open source programming. Despite his best attempts, he remains a regular and voracious programmer on many platforms and numerous environments.

For technical questions or comments about the contents of this tutorial, contact the author, MC, at *questions@mcslp.com* or through his *Web site* (http://www.mcslp.com/contact) , or click Feedback at the top of any panel.

Section 2. Cross-compiling

The need for a cross-compiler

It isn't always possible to write and build an application on the same platform. For many embedded environments, for example, the reduced memory space -- often less than 256 MB, maybe even less than 64 MB, both for RAM and for storage -- just isn't practical. A reasonable C compiler, the associated tools, and the C Library required won't fit into such a small space much less run.

Actually *developing* in such an environment is obviously even more difficult. Niceties like a full-blown editor (for example, emacs) or a full-blown development environment are unavailable, assuming that you can even access and use the system with a keyboard and display. Many embedded solutions don't even have the benefit of network access.

Cross-compilers enable you to develop on one platform (the host) while actually building for an alternative system (the target). The target computer doesn't need to be available: All you need is a compiler that knows how to write machine code for your target platform. Cross-compilers can be useful in other situations, too. I once had to work on a computer with no C compiler installed, and I had no easy way of obtaining a precompiled binary. I did, however, have the necessary source code for the GNU Compiler Collection (GCC), a C library (newlib), and the binary utilities on a computer that did have a C compiler. With these tools, I was able to build first a cross-compiler and then a native compiler for the target computer that I could copy across and use directly.

Cross-compilers can also be handy when you have a slow machine and a much faster one and want to build in minutes rather than hours or days. I've used this method to perform an upgrade to a new software version on a computer on which it would have taken 2-3 days to rebuild all the components -- all while the machine was performing its already-significant server duties.

Before I show you the specifics of building a cross-compiler, let's take a closer look at how compilation works so that you can better understand why -- and more importantly how -- cross-compilation works.

How cross-compilation works

Compilers work in a simple fashion. (In this tutorial, I discuss the GCC, but the basic principles apply to any compiler.) Several different components work together, with the ultimate goal being to produce the bytecode that the target

CPU uses. When you can generate the assembled bytecode, you've successfully cross-compiled.

The main components of any compiler are:

- **Parser:** The parser converts the raw language source code into assembly language. Because you're converting from one format to another (C to assembly), the parser needs to know the target assembly language.
- **Assembler:** The assembler converts the assembly language code into the bytecode that the CPU executes.
- **Linker:** The linker combines the individual object files that the assembler generates into an executable application. Different operating system and CPU combinations generally use different encapsulation mechanisms and standards. The linker needs to know this target format to work.
- **Standard C library:** The core C functions (for example, printf) are provided in a central C library. This library is used in combination with the linker and the source code to produce the final executable, assuming that functions from the C library are used in the application.

In a standard, host-based C compiler, each of these components is designed to produce the corresponding assembly code, bytecode, and target execution format of the host itself. In a cross-compiler, although the application is built to execute on the host, the assembly language, linker, and C library are all designed for the target platform and processor. For example, on an Intel-based Linux machine, you might cross-compile an application so that the assembly language and ultimate application are for a Solaris-based SPARC host.

Building a cross-compiler, therefore, relies on building an alternative version of the C compiler suite that generates and links applications for the target host. Fortunately, because you can compile GCC and the associated tools, you can build your own cross-compilers.

The cross-compiler build process

The GNU utilities (that is, the GCC), including the C compiler, binary utilities, and the C library, have benefits, not the least of which is that they're free, open source, and easily compiled. A much bigger benefit -- from a cross-compiler point of view -- is that because GCC has been ported onto many platforms, the code supports several different CPU and platform types. There are, however, some limitations. GCC doesn't support all processor types (although it produces most), nor does it support all platforms. The configuration tools will warn you about this when executed.

To build a cross-compiler, you need to build three components from the GNU

suite:

- **binutils:** The binutils package includes basic binary utilities such as the assembler, the linker, and associated tools such as Size and Strip. The binary utilities encompass both the core components for building an application and the tools that can be used to build and manipulate the target execution format. For example, the Strip utility removes symbol table, debugging, and other "useless" information from an object file or application, but to do so, the utility needs to know the target format so that it doesn't remove the wrong information.
- **gcc:** The gcc is the main component of the compilation process. Gcc encompasses the C preprocessor (cpp) and the translator, which converts the C code into the target CPU assembly language. Gcc also acts as an interface to the overall process, calling cpp, the translator, the assembler, and the linker accordingly.
- **newlib/glibc:** This library is the standard C library. Newlib was developed through Redhat and can be slightly more user friendly in cross-compilers designed to be used for embedded targets. You can also use the GNU Library (glibc), but I focus on using newlib in this tutorial.

You also need the header files for the target operating system, which are required so that you have access to all the operating system-level functions and system calls that are required to build the application. With Linux, you can get the headers fairly easily. For other operating systems, you can copy an existing set of headers. I look at headers in more detail later.

Optionally, you may want to build the GNU debugger -- gdb -- for the target host. You can't build a debugger that can execute code for the target while running on the host, because doing so would require emulation. However, you can build a gdb executable for your target host.

Section 3. Preparation

Destinations and coexistence

Before you begin the configuration process, you need to determine where the compiler and associated tools will be installed. You have two choices: Either you can install them into a completely separate directory, or you can install them as part of your existing installation.

Among the many benefits of the GNU toolset is the design built into the installation structure, which enables tools and components for different target platforms to coexist. When the software is installed, the installation prefix that you supply is organized according to the normal layout, with the addition of a target directory for the target-specific tools. For example, the structure below is taken from a system on which I've installed a cross-compiler for a generic PowerPC/Linux platform:

```
drwxrwxrwx    2 root     root         4096 Nov 16 16:48 bin/
drwxrwxrwx    2 root     root         4096 Nov 17 12:53 info/
drwxrwxrwx    2 root     root         4096 Nov 17 12:53 lib/
drwxrwxrwx    3 root     root         4096 Nov 16 16:44 man/
drwxrwxrwx    4 root     root         4096 Nov 16 16:48 ppc-linux/
drwxrwxrwx    3 root     root         4096 Nov 16 16:43 share/
```

If you look into the bin directory, you'll see that each of the main binary utilities is prefixed with your build-target:

```
-rwxr-xr-x    1 root     root      2108536 Nov 16 16:46 ppc-linux-addr2line*
-rwxr-xr-x    2 root     root      2157815 Nov 16 16:45 ppc-linux-ar*
-rwxr-xr-x    2 root     root      3398961 Nov 16 16:48 ppc-linux-as*
-rwxr-xr-x    1 root     root      2062804 Nov 16 16:47 ppc-linux-c++filt*
-rwxr-xr-x    2 root     root      2907348 Nov 16 16:48 ppc-linux-ld*
-rwxr-xr-x    2 root     root      2140893 Nov 16 16:46 ppc-linux-nm*
-rwxr-xr-x    1 root     root      2552661 Nov 16 16:46 ppc-linux-objcopy*
-rwxr-xr-x    1 root     root      2708801 Nov 16 16:45 ppc-linux-objdump*
-rwxr-xr-x    2 root     root      2157810 Nov 16 16:46 ppc-linux-ranlib*
-rwxr-xr-x    1 root     root       371010 Nov 16 16:46 ppc-linux-readelf*
-rwxr-xr-x    1 root     root      2008330 Nov 16 16:45 ppc-linux-size*
-rwxr-xr-x    1 root     root      1982880 Nov 16 16:46 ppc-linux-strings*
-rwxr-xr-x    2 root     root      2552660 Nov 16 16:46 ppc-linux-strip*
```

The main tools, such as gcc, are merely wrappers to the background tools that perform the compilation, so gcc can determine which tool to use when building for a different platform. As long as you continue to use gcc for your building requirements and the other libraries and components that you build use the GNU configure structure, you should be able to install your cross-compilation tools alongside your standard toolset. On most Linux platforms, this location is */usr/local.* (I like to keep my cross-compiler and host compiler separate, so that I

can keep different versions of the host and cross-compiler toolsets.)

Identify your target platform

The next step in preparing for cross-compilation is identifying your target platform. Targets within the GNU system have a specific format, and this information is used throughout the build process to identify the correct version of the various tools to use. Thus, when you run GCC with a specific target, GCC looks in your directory path for an application path that includes that target specification.

The format of a GNU target specification is *CPU-PLATFORM-OS.* For example, Solaris 8 for x86 is *i386-pc-solaris2.8,* while Macintosh OS X is *powerpc-apple-darwin7.6.0.* Linux on a PC has the target *i686-pc-linux-gnu.* The *-gnu* tag in this instance indicates that the Linux operating system in question uses a GNU-style environment.

There are many ways of identifying the target, including simply knowing or guessing the target specification. For example, most Linux targets can be specified according to their CPU; hence, PowerPC/Linux is *ppc-linux.* The best method, however, is to use the `config.guess` script, which comes with any GNU suite, including those I use in this tutorial. To use this script, simply run it from a shell:

```
$ config.guess
i686-pc-linux-gnu
$
```

For those systems on which you don't have the ability to run this script, it's worth examining the file to determine some possible targets. Simply using `grep` on your target CPU will give you some idea of the target specification you need. For purposes of this tutorial, I'll create a cross-compiler for the i386-linux platform, a common target and one of the best supported.

Set up your build environment

The final stage before you start building is to create a suitable environment. Doing so is straightforward: You just need to create a simple set of directories that you can use to build the different components.

First, create a build directory:

```
$ mkdir crossbuild
```

Next, obtain the latest versions of gcc, binutils, gdb, and newlib and extract them into the directories:

```
$ bunzip2 -c gcc-3.3.2.tar.bz2|tar xf -
$ bunzip2 -c binutils-2.14.tar.bz2 |tar xf -
$ bunzip2 -c linux-2.6.9.tar.bz2 |tar xf -
$ bunzip2 -c gdb-6.3.tar.bz2|tar xf -
$ bunzip2 -c glibc-2.3.tar.bz2|tar xf -
```

If you're building for a Linux target, you'll also need to unpack the linux-threads package (if your target platform supports it).

Now, create the directories in which you'll actually build the software. The basic layout is to create a single directory for each component:

```
$ mkdir build-binutils build-gcc build-glibc build-gdb
```

If you're creating several different cross-compilers, you might consider creating individual directories for each target, then creating the directories above in each target directory.

With preparation complete, you're ready to configure and build each tool.

Section 4. Configuration and building

Set up the environment variables

Retyping everything is frustrating, and you risk typing the wrong thing. To make life easier, I created several environment variables to save on typing. I've assumed a Bourne-like shell below; if you are using csh, tcsh, or similar shells, you might need to use shell-specific techniques.

```
export TARGET=i386pc
export PREFIX=/usr/local/crossgcc
export TARGET_PREFIX=$PREFIX/$TARGET
export PATH=$PATH:$PREFIX/bin
```

Obtain the operating system headers

The operating system headers are necessary for getting the information the compiler needs for the system function calls that the target platform supports. For Linux targets, the best way to obtain the headers is to download the latest copy of the appropriate kernel. (I used the generic kernel in the previous panel.) You'll need to do a basic configuration of the kernel so that the correct headers are generated for you to copy; you don't, however, need to build or compile the kernel. For my sample target, i386-linux, the necessary steps are:

```
$ cd linux-2.6.9
$ make ARCH=i386 CROSS_COMPILE=i386-linux- menuconfig
```

Note that the trailing hyphen above is not a typo. By running the command above, you'll be prompted to configure the components for the kernel. Because you don't need to worry too much about the contents of the kernel itself, you can just use the default options, save the configuration, and quit the tool.

Now, you need to copy the headers to your target directory:

```
$ mkdir -p $TARGET_PREFIX/include
$ cp -r include/linux $TARGET_PREFIX/include
$ cp -r include/asm-i386 $TARGET_PREFIX/include/asm
$ cp -r include/asm-generic $TARGET_PREFIX/include/
```

Obviously I've made some assumptions in the code above based on the sample target. If you were building for a PowerPC target, you would have copied files from asm-ppc.

You're now ready to start building the utilities.

Build binutils

The binutils utility is the core of the entire system. It provides the basic assembler and linker functionality that the rest of the system requires. The first stage in each case is to configure each package for an alternative target platform. Build binutils first:

```
$ cd build-binutils
$ ../binutils-2.14/configure --target=$TARGET --prefix=$PREFIX --disable-nls -v
$ make all
```

The target and prefix should be obvious. The `--disable-nls` command disables National Language Support (NLS). Many embedded platforms can't support the necessary tables anyway. For most cross-compilers, the usefulness of NLS is debatable because the targets (usually embedded devices) aren't capable of holding the necessary NLS tables in memory.

This stage of the build will take some time. Because you're still building using the host compiler and tools, you can use `ccache` and `distcc` tools to help speed up the process. (For more information about these tools, see the *Resources* (#resources) section at the end of this tutorial.)

Now, you're ready to build GCC, which is slightly more complex.

Build a first-stage GCC

GCC is more complex than binutils only because the standard method for building GCC builds two compilers. GCC uses the GNU tools to build a primary (that is, a first-stage, or *bootstrap*) compiler that can build and parse the basic code. GCC then uses the available library and header files for the target to build the full compiler. Building the GCC first-stage compiler requires a few minor changes to the options for the configuration script so that you can build the first-stage compiler without proper headers. (Strictly speaking, you don't have the headers until you've built a library.) The `--with-newlib` command doesn't necessarily mean you're using newlib: It just tells the configuration script not to worry about headers.

```
$ cd build-gcc
$ ../gcc-3.3.2/configure --target=$TARGET --prefix=$PREFIX \
                         --without-headers --with-newlib -v
```

```
$ make all-gcc
$ make install-gcc
```

Like building binutils, this stage will take a while to complete. The length of time depends on your host, but expect anything from an hour (even on a fast machine) to as many as five or six hours on a slower or busier host.

Building newlib

You can use either newlib of glibc. In general, newlib is better on embedded platforms, because such systems are what newlib was designed to support. Glibc is better for Linux-style hosts.

When building newlib, you need to build the library using the target compiler and tools. The library should, of course, be in the format and language of the target CPU and platform for it to be used to build applications that rely on the library components:

```
$ cd build-newlib
$ CC=${TARGET}-gcc ../newlib-1.12.0/configure --host=$TARGET --prefix=$PREFIX
$ make all
$ make install
```

When newlib has been built, you can create the final GCC based on this code to create the final compiler. Alternatively, you can use glibc, which I cover in the next panel.

Build glibc

The glibc package is straightforward to build; the primary difference from the previous builds is that, as with newlib, you now start to use the bootstrap cross-compiler you just built. You also need to tell the `configure` script where the header files for the operating system are kept. Finally -- and here's the big difference -- you define the host you're building on rather than the target. That's because the GCC and binary utilities you've already built mean that this machine is your development host; the GCC you specify will generate the necessary target code for you.:

```
$ CC=${TARGET}-gcc ../glibc-2.3/configure --target=$TARGET \
                        --prefix=$PREFIX --with-headers=${TARGET_PREFIX}/include
$ make all
```

If you wanted to include the Linux threads option, you need to add the

`--enable-add-ons` option to the `configure` script. Again, this process will take some time to complete. Nobody ever said building cross-compilers was quick!

To install glibc, you still use `make`, but you explicitly set the installation root and empty the prefix (otherwise, the two are concatenated, which is not what you want):

```
$ make install_root=${TARGET_PREFIX} prefix="" install
```

Finally, you can build the final version of gcc, which now uses the above library and header information.

Build the final GCC

The final GCC makes use of the headers and libraries that you've just compiled (using your chosen target) to build the full gcc system. Unfortunately, this means an even longer wait while the full version of gcc is built. I build only a C compiler with gcc rather than building the full suite.

You don't need to worry about the old gcc build, so you can remove that content and start again in the build-gcc directory. The configuration is just like the previous examples. Note that because you're building a tool that will be executed on this platform, you're back to using the host GCC compiler rather than the bootstrap GCC compiler you built earlier:

```
$ cd build-gcc
$ rm -rf *
$ ../gcc-3.3.2/configure --enable-languages=c --target=$TARGET --prefix=$PREFIX
$ make all
$ make install
```

Because you're building with the full newlib and gcc components for the target platform, using `distcc` is not possible without also installing those components on the other machines in your distcc host list. Even with distcc, it's going to take some time, even on a fast machine. I tend to leave these builds running overnight if at all possible -- partly because of the time, but partly because of the increased load it puts on the build machine (or machines), which can be annoying if your machine is used for other purposes at the same time.

When the build is finished, though, you're ready to start using your cross-compiler to build other the applications and libraries you need for your target platform.

Section 5. Install and use your cross-compiler

Use your cross-compiler

Actually using the cross-compiler is easy. To compile a simple C file, just call the cross-compiler directly:

```
$ i386-linux-gcc myapp.c -o myapp
```

The GCC cross-compiler works just like your local version: It just creates a different type of executable for an alternate platform. This means that you can use the same command-line options, such as header and library locations, optimization, and debugging. (Remember, however, that you can't link libraries for different target platforms together and expect them to work.) In the next panel, I show you how to build libraries and extensions with your new cross-compiler.

For applications and projects where you have a Makefile, specify the cross-compiler on the command line to `make`. For example:

```
$ make CC=i386-linux-gcc
```

Alternatively, change the CC definition within your Makefile.

Compiling other tools and libraries

Any libraries you might want to use on your target platform need to be compiled with the cross-compiler for them to work. For most libraries, the same basic rules apply as for building your own projects. If a system uses a simple Makefile, use:

```
$ make CC=i386-linux-gcc
```

Or, if you're using a configure script, prefix the `configure` command with the re-definition of the CC environment variable:

```
CC=i386-linux-gcc ./configure
```

With GNU-style configure scripts, you may also need to specify the host:

```
$ ./configure --host=i386-linux
```

Remember that for libraries, you need to specify the target prefix, just as you did when building glibc or newlib.

Section 6. Summary and resources

Summary

Building a cross-compiler is certainly a time-consuming task, and in some ways a complicated one. But when you get over the basic needs of setting targets and the sequence, creating the cross-compiler is relatively straightforward.

The benefits, especially when developing for an embedded platform, are immeasurable. With some systems, it's simply not possible to generate the build locally, and running gcc on a system with just 64 KB is well nigh impossible. With a cross-compiler, you can develop on your preferred platform with your preferred tools and environment and still end up with the application and software you need.

Resources

- You can get copies of all the source code for the GNU tools from *GNU* (http://www.gnu.org) . Please use a local mirror to help reduce the load on the GNU servers.
- You can obtain newlib, the embedded alternative C library, from *Redhat Sources* (http://sources.redhat.com/newlib/) . There's also a *Mailing list* (http://sources.redhat.com/newlib/mailing.html) .
- The *CrossGCC Mailing List Archives* (http://sources.redhat.com/ml/crossgcc/) can be useful if you have an issue with a specific target or build environment.
- The *Crossgcc Wiki* contains all sorts of useful information about building cross-compilers.
- The *ccache* and *distcc* can help to improve the build times of applications, including the early stages of a cross-compiler.
- For a very brief overview of the sequence, try the *Mini How-To* (http://linux.bytesex.org/cross-compiler.html)
- The developerWorks *Migration station Linux section* offers a wealth of links to information for those interested in porting and migrating applications to Linux.
- Find more resources for Linux developers in the *developerWorks Linux zone* (http://www.ibm.com/developerworks/linux/) .
- Get involved in the developerWorks community by participating in *developerWorks blogs* (http://www.ibm.com/developerworks/blogs/) .
- Purchase *Linux books at discounted prices* in the Linux section of the

Developer Bookstore.

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like developerWorks to cover.

For questions about the content of this tutorial, contact the author, Martin C. Brown, at *questions@mcslp.com* or through his *Web site* (http://www.mcslp.com/contact) .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .