CHAPTER 6



# Creating a Linux Distribution from Scratch

Creating a Linux distribution from scratch is viewed as a daunting task, and it shouldn't be. The process involves creating a toolchain, using that toolchain to create a root file system, and then building a kernel. The process isn't complex, it just requires you to carefully follow many steps to build the proper tools. After you go over the instructions for building a toolchain and understand how it works, the smartest route may be to use a tool that builds a toolchain, such as crosstool-NG, to build the root file system and kernel.

In prior years, before Linux's popularity with board vendors and chip makers hit its stride, support for any hardware platform other than Intel's X86 was spotty; however, with Linux becoming the platform of choice for embedded devices, support for most embedded hardware platforms is part of the Linux kernel and toolchain.

Chapter 5 discussed getting a Linux distribution from another source, such as the vendor that sold you the board, a commercial vendor, or one of the several open source distribution building project. These are fine approaches to getting a Linux distribution, but building from scratch gives you the most control over the composition of the distribution and how it works. The learning curve to create a distribution from scratch is also about the same (at least for me) as learning a distribution-building tool, and the skill is more transferable.

Making a Linux distribution involves these steps

1. Build a cross-compiler.
2. Use the cross-compiler to build a kernel.
3. Use the cross-compiler to build a root file system.
4. Roll the root file system into something the kernel can use to boot.

A cross-compiler is the Rosetta stone (the flux capacitor, if you will) of embedded development. Until this tool has been built, the kernel can't be built—nor can the root file system's programs be built.

This chapter goes over how to create a toolchain for an ARM board and then build the kernel and root file system. Because the Linux kernel has matured, the sources in the main projects don't need the level of patches they once did in order to support common embedded targets The hunting and gathering step snares many who build distributions from scratch, because there is no canonical place where you can locate the patches for a certain revision of the tools used in the build.

Building the toolchain is still one of the more complex bits of the process, and a reasonable tool is available that just builds a toolchain: crosstool-NG. After you build a toolchain from scratch, the chapter has you create one with crosstool-NG to compare the two processes. If possible, the best approach is to use crosstool-NG or one of the distribution builders to create a toolchain. Because a toolchain involves so many software packages, getting a set that works together is a great value that these projects provide.

Another slightly confusing part of the process is that the kernel sources are used in the build process. One step in building the cross-compiler is to build the corresponding C Standard Library. In order for the C Standard Library to build, it needs to know some information about the target machine, which is kept in the kernel. For the uninitiated, this seems to be a circular dependency; but the parts of the kernel that are used don't need to be cross-compiled.

When you have the cross-compiler in hand, you create the root file system with the BusyBox project. Linux requires a root file system and refuses to boot if one can't be found. A root file system can be one file: the application program for the device. Most embedded Linux systems use additional libraries and utilities. The root file system in this chapter is linked with the GNU C Library at runtime, so these files must be on the target as well.

BusyBox is a program that provides limited-functionality implementations of most command-line tools found on a Linux system. Although a root file system doesn't need to contain all these tools, having them available when the system boots is a convenience. When you understand what tools are necessary (frequently, it's a very small set), you can remove the rest to economize on space and make the system more secure.

How does a small root file system make a system more secure? The fewer programs, the smaller number of possible exploits. A root file system containing just the functionality necessary to execute the application on the target doesn't present an attacker with any more opportunities than absolutely necessary. A small number of programs also reduces the maintenance required to keep current with changes happening in the open source community.

The kernel is the last component in this process. The kernel is a self-contained system that doesn't have runtime dependencies. Being a very mature project, the cross-build of the kernel works very well. The kernel build process is by far the easiest process in this chapter.

# Cross-Compiler Basics

A *cross-compiler* is a tool that transforms source code into object code that will run on a machine other than the one where the compilation was executed. When you're working with languages that execute on virtual machines (like Java), all compilation is cross-compilation: the machine where the compilation runs is always different than the machine running the code. The concept is simple in that when the compiler generates the machine code what will eventually be executed, that code won't run on the machine that's doing the generating.

Architecturally, from a compiler designer's perspective, the target machine is a few steps removed from the code-generation process, because the intermediary steps in compilation produce output that's designed to be easy to optimize rather than run on hardware. After the optimization steps, the compiler then produces code intended to run on an actual processor. Thinking about the compiler's operation in this way makes it easy to understand that the final step of turning the optimized pseudocode into the machine code for the target can produce arbitrary machine code and not just code that executes on the host machine.

In this section, the focus is on creating a GNU GCC cross-compiler as the first step to creating an embedded Linux platform. First you build the supporting binutils, then a cross-compiler suitable for compiling glibc, and then the final compiler. For the purpose of illustration, the steps are broken out into several sections. In a real project, all the steps are combined into a script that can be run without intervention.

Some basic terminology is used to describe the players in the process of building the compiler:

- *Build machine:* The computer used to compile the code
- *Host machine:* The computer where the compiler runs
- *Target machine:* The computer for which GCC produces code

The concept of *target* throws many people for a loop. Table 6-1 shows two examples that should make clear the distinction between build, host, and target machines. This table also contains the Canadian cross-compiler case, which is explained in the following paragraph.

***Table 6-1.*** *Standard Designations for Cross-Compilers*

| Machine | Standard Cross-Compiler | Canadian Cross-Compiler |
|---|---|---|
| Build | Linux x86 | Linux x86 |
| Host | Linux x86 | Apple MacOS X86 |
| Target | Linux ARMv7l | Linux ARMv7l |

A *Canadian*[1] cross-compiler build is one where the build, host, and target machines are all different. You probably aren't configuring the compiler to run on anything other than an x86 Linux host, but you may need to support development groups running on Windows or Mac OS and want to build the software on a Linux host. It's possible that the build is running an a 64-bit Linux host, producing a compiler that runs on a 32-bit host that generates code for an ARM processor, so it pays to understand the mechanics of cross-building.

The notion of a Canadian cross-build is generally reserved for compiler building. For most other projects, the build and host machines are equal, and only the target changes.

## A Note about Building Software

Building open source software like the toolchain involves using a piece of software called Autoconf. This software (which itself is quite complex) creates a make file after running a script called `configure` that inspects the system where the build occurs. When running `configure`, you pass in parameters describing how to build the software for which the `configure` script can't use a reasonable default.

Some of the most confusing parameters passed into `configure` are the values for host, target, and build. These parameters are called *configuration names*, or *triplets*, despite the fact that they now contain four parts. When you're building a toolchain, the triplet describing the target machine is the four-part variety: for example, `arm-none-linux-gnueabi`. This unwieldy string prefixes the name of the toolchain executables so it's clear what the cross-compiler is targeting. It's possible to change this prefix, but the best practice is to leave it be.

Table 6-2 shows a breakdown of the different parts of the configuration name.

[1] This describes a certain parliamentary configuration in Ottawa. I had a Canadian buddy explain what this meant, but I lost interest after about the first 30 seconds and pretended to listen for the remaining time before excusing myself to use the bathroom. Canadian politics aren't as interesting as I expected.

***Table 6-2.*** *Configuration Name Values*

| Part | Description | Common Values |
|---|---|---|
| CPU | The processor label. Use the manufacturer of the CPU, not the board where the CPU resides. | `arm`, `mips`, `i686`, `i486`, `x86_64`, `ppc`, `powerpc`. |
| Manufacturer | This can be any value; it's here for informational purposes. For example, the ARM CPU design is built by any number of manufactures, so this could be the ARM licensee that built the silicon. | Free form. `none` is the most popular. `pc` is a common value for Intel systems. |
| Operating System | Operating system | Linux. |
| Kernel | This describes the libc running on the system. When you're building a toolchain, use `gnuabi` as the value. | `gnuabi`, `gnueabi`, `gnulibc`, `uclibc`. |

Many users get flustered about what to pick for their processor. The acceptable values for this field are always in flux, and experimentation is a reasonable approach to verify that the triplet is acceptable.

Another thing that is different from other open source packages is that the glibc and libc configuration steps must be done from a directory other than one where the source code files reside. Although the directory can be any one other than where the source resides, the best practice is to create an empty directory and run the configuration program from the newly created directory. One of the great advantages of this practice is that creating a clean build environment is just an `rm -rf <build directory>` away. Given the experimental nature of building toolchains, being able to start from scratch when building is a great convenience.

## Get Comfortable with the Command Line

The examples in this chapter are all executed from a command-line environment. The engineers who build and maintain the kernel, compiler, and libraries work from the command-line, because it's the *lingua franca* of the Linux world. This doesn't show a bias against using IDE tools; years, and in some cases decades, of time and effort have gone into making the command line efficient and productive. After these tools have been built, several graphical environments know how to use them, so time spent at the command prompt is minimized.

These examples use the bash shell. Nearly all distributions use this shell by default. The vast majority of users have graphical desktops, and you run the shell by selecting it from the menu; for example, in Ubuntu, you can start the shell by selecting Applications ➤ Accessories ➤ Terminal from the main menu. If you can't locate the shell on the menu, press the Alt+F2 key combination. On both Gnome and KDE, you're prompted for a program to run (see Figure 6-1; you want to use `/bin/bash`. Also, be sure to select the "Run in terminal"[2] check box (or the equivalent for your window manager); otherwise, you won't see your shell.

[2] It seems odd that you must select "Run in terminal" to … run a terminal, but that's what necessary. It seems to be tempting a total protonic reversal.

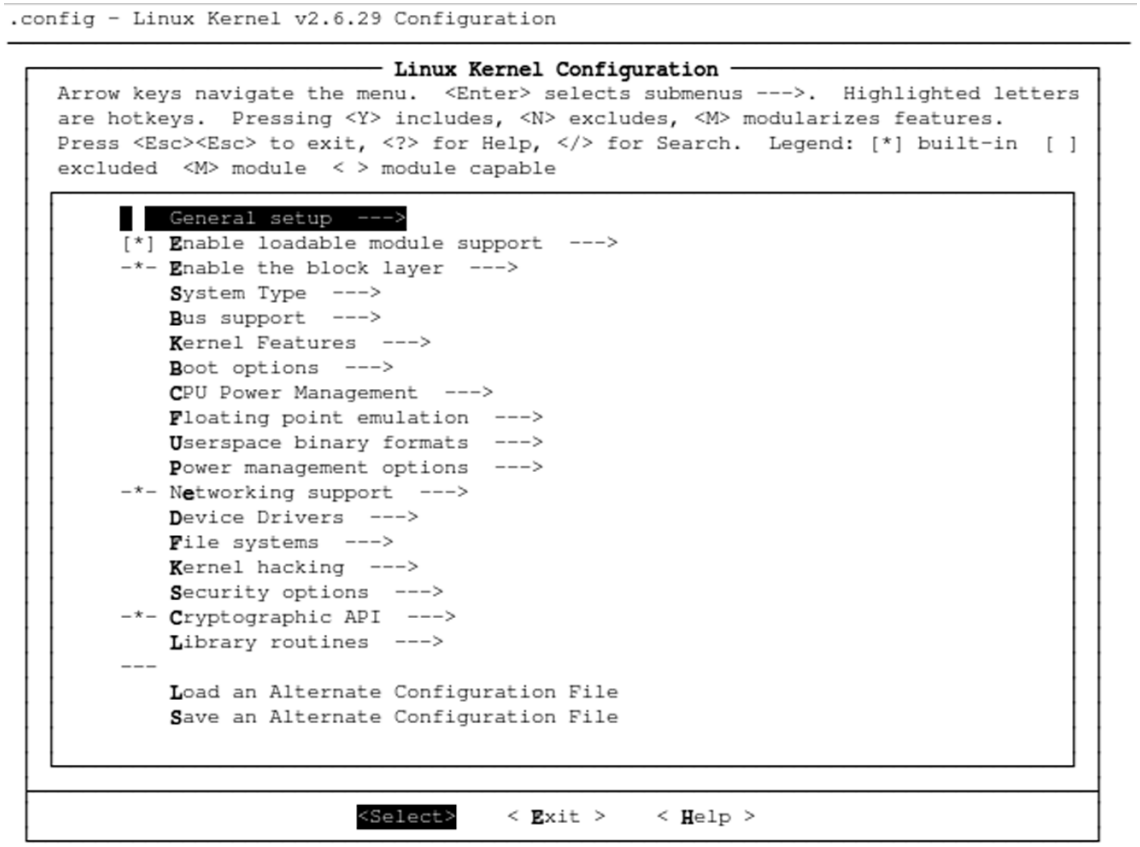


***Figure 6-1.*** *Run Application dialogs*

If you're new to the command line, a great hint is to open several command-line terminal windows. This makes it easy to experiment in one window and use another check on the results. A terminal window requires very few resources, so there's no need to economize on the number open at any one point as long as confusion is kept at bay.

# Overview of Building a GCC Cross-Compiler

Building GCC involves fetching the sources for several projects, getting them ready for compilation, and then running the compile. The entire process with a reasonably fast computer takes an hour or and requires about 500GB of disk space. In the process, GCC is built twice. An overview of the steps follows:

1. Gather the sources. There are two ways to get the source code for a project: get the sources from version control, or fetch source snapshots created when the software is released. If you're planning to contribute to the development efforts of the projects used in the construction of a toolchain, using source controls is the route you should take; otherwise, you should use the source snapshots. No matter how you plan to get the sources, fetching the source code over a slow connection can sometimes require an hour.
2. Build the binutils. The binutils (binary utilities) provide low-level handling of binary files, such as linking, assembling, and parsing ELF files. The GCC compiler depends on these tools to create an executable, because it generates object files that binutils assemble into an executable image.
3. Build a bootstrap GCC. You build GCC once with minimal settings so that it can be used to then build the C Standard Library—in this case, GNU C Standard Library (glibc). The bootstrap compiler has no such library: it can be used to compile C code, but the functions that most C programmers depend on to interact with the underlying machine, such as opening a file or printing output, don't exist for this compiler.
4. Use the bootstrap GCC to build the glibc library. With the C library, you have a tool you can use to build an executable that runs on the target system. It has the code that opens files, reads, writes, and otherwise functions as you expect when you're using C.

## The C Library

In this chapter you build a Linux toolchain with the GNU C Library. This is the library running on the vast majority of desktop Linux machines, and it's the most complete implementation of the C Library for Linux systems. You may even say it's the canonical implementation. Other options for the C Library that are commonly used in embedded systems are addressed later in the book. The process for building a toolchain that uses a different library, like dietlibc, newlib, or uClibc, follows the same pattern of creating a bootstrap compiler, then building the libraries, and then rebuilding the compiler.

The GNU C Library (glibc) is the most ornery to build and get working. After you pass the glibc trial, getting newlib or another library built is much easier. Later in the book, the libraries are discussed and compared to help you select the best one for your project.

## Gathering Sources

There are two ways to get the sources: download distribution tars or get the source directly from source control. If you're planning to participate in the development of these projects, get the files from source control, because this is the easiest way to contribute patches and stay current with the development process. If you're planning to use the sources to make a build, and you aren't interested in contributing to the projects, skip the section where the files are fetched as tar archives.

Before you download a bunch of open source projects in your home directory, create a new directory to hold these files so you can stay organized. This directory is known as the *source directory* and is referred to as `$SRCDIR` later in the chapter:

```
$ export $SRCDIR=~/xtools/src
$ mkdir -p $SRCDIR
```

## Getting Sources via Source Control

A *toolchain* is a collection of tools that work together. Each of these tools is a separate project that is maintained by a different group of people. Each project has selected a version-control system and a source-management system that work best for it, and that means not all the projects use the same systems. The following sections outline the commands to fetch the code necessary to build a toolchain from the various projects.

If you don't plan to develop this code, or you don't need the most recent version, skip ahead to the next section, where the release snapshots are downloaded.

### Binutils

Binutils is stored in a CVS repository that you can access as an anonymous user by doing the following:

```
$ cvs -z 9 -d :pserver:anoncvs@sourceware.org:/cvs/src login

(use anonymous when prompted for the password)
$  cvs -z 9 -d :pserver:anoncvs@sourceware.org:/cvs/src co binutils
```

As an anonymous user, you can download the code, get updates, and, if you make changes, easily generate diff files that you can send to the project maintainer for inclusion into the project.

### GCC

The GCC sources are kept in a subversion source control system. Subversion (SVN) is a successor to Concurrent Versions System (CVS) and shares many of the same commands and semantics of CVS. To get a project, you perform a checkout; when you want to take a snapshot of the source code—say, to get the state of the files for a release—you can tag the sources. The implementation is very different than CVS and in many cases an improvement.

Each release of GCC is tagged with the format `gcc_`*`A`*`_`*`B`*`_`*`C`*`_release`, where *A*, *B*, and *C* are the major, minor, and revision numbers, respectively. Tags in SVN pin the version of the file at a certain point; every time a tag is downloaded, the files are the same. To get a listing of the release tags for the GCC project, do the following:

```
$ svn ls svn://gcc.gnu.org/svn/gcc/tags | grep _release/$
(clipped)
gcc_4_2_4_release/
gcc_4_3_0_release/
gcc_4_3_1_release/
gcc_4_3_2_release/
gcc_4_3_3_release/
```

This command says, "List all the files in the `tags` directory, and filter to find the lines ending in `release/`." The tag tree contains many other entries, as you can imagine for a complex project like a compiler. To see a complete listing of the tags, drop the `grep` filter at the end of the line. To identify the latest release, visit http://gcc.gnu.org and look at the Status section; the version number of the release is clearly displayed as a link to a page with more detail about the release and a link for downloading the source.

There is also a branch where active work occurs for each version, formatted `gcc-`*`A`*`_`*`B`*`-branch`: *A* and *B* are the major and minor versions, respectively. A branch's files, unlike a tag's, aren't immutable. Use a branch to stay abreast of the changes in a certain version of GCC; use a tag if your goal is to get the same

files for each checkout, which may be necessary to create a repeatable build process. The branch where the newest work goes is the *trunk* branch. All the newest, not yet officially released code goes into the trunk branch; when it's ready, the trunk becomes the next release.

To get a listing of the current branches, do the following

```
$ svn ls svn://gcc.gnu.org/svn/gcc/branches | grep  "gcc-*[0-9]_*[0-9]-branch"
gcc-3_4-branch/
gcc-4_0-branch/
gcc-4_1-branch/
gcc-4_2-branch/
gcc-4_3-branch/
```

If you're more comfortable getting the sources the old-fashioned way, they're still available as tar files. The easiest way to fetch a tar is using `wget`, which connects to a remote server and fetches the URL.

## Glibc

Next, get glibc. This is a little more complex because in order get support for the ARM architecture, you must download two things: glibc and the special ports project. Some architectures aren't supported directly by the glibc team, with ARM being one of them:

```
$ cvs -z 9 -d :pserver:anoncvs@sources.redhat.com:/cvs/glibc login
$ cvs -z 9 -d :pserver:anoncvs@sources.redhat.com:/cvs/glibc co -r glibc-2_9-branch
libc
$ cd libc
$ cvs -z 9 -d :pserver:anoncvs@sources.redhat.com:/cvs/glibc co -r glibc-2_9-branch
ports
```

## GMP and MPFR

The Gnu Multiple Precision (GMP) library uses Mercurial for source code management. Most systems don't have this installed by default; you can install it through the package-management system. Use this for Debian-based systems:

```
$ apt-get install mecurial
```

Or use this for RPM systems:

```
$ yum install mercurial
```

After Mercurial is installed, you check out a project much as in other source code control systems:

```
hg clone http://gmplib.org:8000/gmp
```

This command checks out the sources of the project into the `gmp` directory. Mercurial is a distributed source control system like Git and works the same, in principle.

MPFR uses SVN for source control. You access the repository with the following command:

```
svn checkout svn://scm.gforge.inria.fr/svn/mpfr/trunk mpfr
```

### A Linux Kernel

The Linux kernel project is stored as a Git repository. To get this from source control, you need to clone Linus's Git tree by doing the following

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

This tree has quite a bit of data, and the cloning process can take upwards of 45 minutes on a slow Internet day. The GCC (and most other GNU sources) are available for download from GNU's FTP server at `ftp://gcc.gnu.org/pub`. In the case of GCC, the project is located at `ftp://gcc.gnu.org/pub/gcc/releases/gcc-<release>`. You can navigate this FTP server using your browser: just type the URL into the web address control. For example, to download the GCC 4.3.3 source tarball, do the following:

```
wget ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-4.3.3.tar.bz2
```

This command fetches the file at `ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-4.3.3.tar.bz2` and puts it in into the current directory as `gcc-4.3.3.tar.bz2`.

---

■ **Note** `bz2` or `gz`? In this chapter (as well as the rest of the book), open source software is frequently offered as both `.tar.gz` and `tar.bz2` files. Both of these extensions are compression systems. The newer `bz2` compression algorithm does a better job making a smaller file, but `gz` is more widespread. In both cases, the compressed file is in tar format, which is supported by every system.

---

## Getting via Source Archives

You'll fetch files from a variety of places. Although the examples have specific version numbers, there will probably be something newer by the time this book makes it into your hands. These files are kept at FTP servers, and you can easily download them using `wget`. Following is a brief section describing how to get the sources for each. After you download the files, you can uncompress them with tar.

### Binutils

This archive is fairly small and downloads quickly. For this example, you use version 2.20. If you want a different version, change the name of the file to the version number to download. If you're curious about what versions are available, point your browser to `http://ftp.gnu.org/gnu/binutils` for a directory listing:

```
$ wget http://ftp.gnu.org/gnu/binutils/binutils-2.20.tar.gz
```

Then, unpack the file using the following command.

```
$ tar xzf binutils-2.20.tar.gz
```

### Glibc

Glibc is a little more complex, because you must download both the sources for the project and the add-ons, and you need to uncompress the add-ons into the glibc directory:

```
$ wget ftp://ftp.gnu.org/gnu/glibc/glibc-2.9.tar.bz2
$ wget ftp://ftp.gnu.org/gnu/glibc/glibc-ports-2.9.tar.bz2
```

Unpack the files as follows:

```
$ tar xjf glibc-2.9.tar.gz
$ cd glibc-2.9
$ tar xjf ../glibc-ports-2.9.tar.bz2
```

---

■ **Note** The `ports` code is necessary only if glibc is being built for the following architectures: MIPS, ARM, and PowerPC. Other architectures are supported in the main glibc project.

---

### GCC

The source code is located at `ftp://gcc.gnu.org/pub/gcc/releases`, where you can pick your release tar file. This example uses 4.3.3; you can download this file by doing the following:

```
$ wget ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-4.3.3.tar.gz
```

Each GCC release is in its own directory, where there are archive files for each release and a test suite. In this example, you download the tar file with the complete GCC sources.

### GMP and MPFR

Like the other projects, these projects create source snapshots for each release. You get both with these commands:

```
$ wget http://www.mpfr.org/mpfr-current/mpfr-2.4.1.tar.bz2
$ wget ftp://ftp.gmplib.org/pub/gmp-4.3.1/gmp-4.3.1.tar.bz2
```

Now you have all the files necessary to build GCC.

### A Linux Kernel

You need not just any random Linux kernel, but the one you'll be using on the board. The build process uses some of the headers from the kernel sources to build the compiler and the standard library. This doesn't mean the standard library has to be rebuilt when you switch kernels, because the software interfaces presented by the kernel are stable at this point. Here, you fetch Linux kernel 2.6.28; you should get the kernel that you plan to use with your board:

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.28.3.tar.bz2
$ tar xjf linux-2.6.28.3.tar.bz2
```

## Building GCC

The build process for this (and most other) open source software follows the configure/make/install pattern to build the source code. The configure step runs a script that inspects the system, ensuring that the necessary tools are present and interrogating the compiler's support features. The result of the configuration script is a file with the system settings, files necessary to build the project, and a make file. The make file is a complex but standard GNU make file and is run from the command line to actually build the software. When that's complete, you use make again to install the software.

---

■ **Note** During the process of building a toolchain and libraries, you may want to experiment with the options you send to `configure` to get a different toolchain for your specific target architecture. Experimentation is good. Always start with an empty build directory! The debris from the prior `configure` can result in some very unpredictable problems. Because the configuration occurs in a separate directory from the source code, you can unceremoniously delete the contents with `rm -rf *`.

---

None of these packages let you build from the source directory, so you need to create another directory for the purpose of running `configure` and running the build. You may find this confusing at first, because the directory where make runs doesn't have any files to make, such as in other projects. But rest assured, the projects will build.

## The Build Environment

After you download all those files, it's easy to get confused about what is where. In this example, all the source files are in `$SRCDIR`, with a directory for each of the projects. So that this section is easier to follow, it introduces some additional environment variables to make the examples clearer (see Table 6-3). Because the source code may have been downloaded from release snapshots or checked out from version control, the directory on your system will be different for each.

***Table 6-3.*** *Environment Variables Used for Toolchain Build*

| Shell Variable | Definition |
|---|---|
| `$SRCDIR`<br>The base source directory. All the sources for the build should reside underneath this directory. | `~/xtools/src` |
| `$BINUTILS_SRC`<br>The sources for the binutils project. | `$SRCDIR/<`*`binutils source directory`*`>` |
| `$KERNEL_SRC`<br>The sources for the kernel. | Location of the kernel used with the target |

| | |
|---|---|
| `$GCC_SRC`<br>The sources for GCC. | `$SRCDIR/<gcc source directory>` |
| `$GLIBC_SRC`<br>The sources for glibc. The decompressed ports archive is in this directory. | `$SRCDIR/<glibc source directory>` |
| `$GMP_SRC`<br>The sources for GMP. | `$SRCDIR/<gmp source directory>` |
| `$MPFR_SRC`<br>The sources for MPFR. | `$SRCDIR/<mpfr source directory>` |
| `$BUILDDIR`<br>The directory where the binaries are built. | `~/xtools/build` |
| `$TARGETMACH`<br>The machine that is executing the code built by the toolchain. | The triplet for the target machine. In this case, it's `arm-none-linux-gnueabi`. |
| `$BUILDMACH`<br>The machine where the build is running. | Set this to `i686-pc-linux-gnu`. The configure scripts usually guess the right value for this field; specify it here to be pedantic. |
| `$INSTALLDIR`<br>Where the toolchain is installed on the build machine. | Where the toolchain binaries are installed. This can be any directory. You need write privileges to this directory. In this example, use `/opt/arm`. |
| `$SYSROOTDIR`<br>The location where the target system's libraries and header files belong. | Needs to be under the toolchain installation directory. It can be any name that doesn't conflict with a directory that the toolchain creates. In this example, use `$INSTALLDIR/sysroot`. |

For each of the steps, a little code deletes the prior build directory, if it exists, and re-creates the build directory. This ensures that the configuration and build environment are completely clean and aren't affected by prior builds:

```
[ -d $BUILDDIR/<build directory> ] \
          && rm -rf $BUILDDIR/<build directory>
mkdir $BUILDDIR/<build directory>
```

The first line performs an `rm -rf` if the directory already exists; otherwise, the command is skipped. The next line creates the directory so that the output of the `configure` script has someplace to go.

## Binutils

The first project to build is binutils, which supplies common routines for handling Executable and Linkable Format (ELF) based executable files. ELF is the format that Linux uses for binary executable files. The GCC compiler relies on binutils to provide much of the platform-specific functionality. Binutils provides the following utilities:

- `as`: The Gnu assembler. Transforms mnemonic assembler code into opcodes for the target processor.
- `ld`: The linker. It takes as input a group of object files and creates a binary file such as an executable or shared library or other object file. The linker lays out the file in its final format.
- `ar`: The archive program that packages a group of object files into a single library file to make using the object files more convenient.
- `objdump`: Prints out the contents of an ELF file. Can be used as a disassembler.
- `objcopy`: Makes a copy of an ELF file, allowing you to control what is copied in ELF terms, such as selecting certain sections of the file or removing debugging symbols.
- `c++filt`: Demangles a C++ symbol. The compiler generates internal symbols to handle classes, namespaces, and overloaded functions; this undoes the process to make finding problems in source code easier.
- `size`: Prints the size of sections in the file. This is an important tool that's used when minimizing a system's size.
- `strings`: Prints the null-terminated string constants in a program.
- `strip`: Removes debugging and other bookkeeping information.

Building binutils is a nice place to begin building a toolchain, because it's very straightforward and rarely fails. Look at Table 6-4 to analyze the build options:

```
$ [ -d $BUILDDIR/binutils ] && rm -rf $BUILDDIR/$binutils
$ mkdir $BUILDDIR/binutils
$ cd $BUILDDIR/binutils

$ $BINUTILS_SRC/configure  \
        --disable-werror \
        --build=$BUILDMACH \
        --target=$TARGETMACH \
        --prefix=$INSTALLDIR \
        --with-sysroot=$SYSROOTDIR
```

***Table 6-4.*** *Build Values and their Explanations*

| Configure Parameter | Meaning |
|---|---|
| `--build = $BUILDMACH` | The type of build system. The `configure` script can guess this, but it's a good habit to tell it what system is being used. |
| `--target=$TARGETMACH` | What machine is being targeted for the build. The value here follows the same format as the `build` parameter. In this case, you're building for an ARM processor that is running Linux. |
| `--prefix=$INSTALLDIR` | The installation directory of binutils. This is called `prefix` because it's tacked on to the front of the normal installation directory. |
| `--disable-werror` | The compiler builds with the flag that considers warnings as errors. This version has a few warnings; using this flag allows the build to continue running in spite of the one or two warnings that appear. |

The `configure` script runs and produces output showing what it's inspecting. When it's complete, run the make:

```
$ cd $BUILDDIR/binutils
$ make
```

The software builds; this takes about 20 minutes on a typical machine. When the build completes, put the newly built binutils in the installation directory:

```
$ make install
```

## Kernel Headers

The next step is getting the headers for the kernel that is used on the target board. This involves using the kernel sources to create some header files and then copying those header files into a directory that the other tools will use. This may seem a bit confusing, but these files are necessary because the kernel has the machine-specific information (such as the number of bytes in an integer) that the remainder of the programs need to build code for the target. Furthermore, these files have information about how to call kernel services that the C libraries use to access kernel functionality, because much of the C Standard Library is a layer over the operating system, providing a uniform API or accessing files for the heap.

To get these files, the kernel must be configured for the target machine. In this case, the target is the ARM Integrator board. This is a board that ARM, Ltd. uses in its developer kits and is therefore well supported in the kernel.

Unlike other steps, this step uses the `make mrproper` command to clean up the environment before the build. Running this command puts the kernel source directory in the same state as when the directory was unpacked:

```
$ cd $KERNEL_SRC
$ make mrproper

$ make ARCH=$TARGETARCH integrator_defconfig
```

The screen fills with output as the system tells you that it's first building some tools that will run on the host platform and then sets the configuration for the kernel build to the settings in the `integrator_defconfig` file. The *integrator* is a device created by ARM for doing development work and is the closest thing in the ARM architecture to a generic device. If you have a specific ARM board in mind for your project, you can select that board's `defconfig` now. Of course, if you're building for a non-ARM architecture, select the appropriate `defconfig` for your board. A `defconfig` is the kernel's way of storing the default set of build settings and is discussed in "Building the Kernel" section later in this chapter.

If you're new to building the kernel (also covered in detail later in the chapter), the first step is putting the source tree in the right state to build for the target. It's not necessary at this point to configure the particulars of the kernel; you just need to get the architecture and processor correct.

Now that the kernel source tree is ready to build for the intended target, this command results in the proper headers being generated and then installs them to the toolchain:

```
mkdir -p $INSTALLDIR/sysroot/usr
    make ARCH=$TARGETARCH headers_check
    make ARCH=$TARGETARCH \
INSTALL_HDR_PATH=$INSTALLDIR/sysroot/usr \
headers_install
(clipped)
  INSTALL include/linux/wimax (1 file)
  INSTALL include/linux (349 files)
  INSTALL include/mtd (6 files)
  INSTALL include/rdma (1 file)
  INSTALL include/sound (9 files)
  INSTALL include/video (3 files)
  INSTALL include (0 file)
  INSTALL include/asm (31 files)
```

## Bootstrap (Stage 1) GCC

You're ready to build the bootstrap GCC. Building GCC for a production system is done in several steps, where each steps builds a compiler with more capabilities. The first step is creating a compiler that can then be used to build the Standard C Library (in this case, glibc, but it could be another library). The compiler without a Standard Library is functional, but it doesn't have the bits necessary to produce an executable on a Linux platform. You replace the bootstrap compiler with the final compiler later in the process.

---

■ **Note** Although a compiler without the standard libraries isn't that useful for producing applications, it's a perfectly good tool for building a Linux kernel, which itself has no dependencies on a Standard C Library.

---

The build for this is more complex than for prior packages, because you're sending more information into the `configure` script and the make targets aren't standard. In addition, after running the build, you need to create a symlink so that other components can build properly:

```
[ -d $BUILDDIR/bootstrap-gcc ] && rm -rf $BUILDDIR/bootstrap-gcc

mkdir $BUILDDIR/bootstrap-gcc
cd $BUILDDIR/bootstrap-gcc

$SRCDIR/$GCC/configure \
    --build=$BUILDMACH \
    --host=$BUILDMACH \
    --target=$TARGETMACH \
    --prefix=$INSTALLDIR \
    --without-headers \
    --enable-boostrap \
    --enable-languages="c" \
    --disable-threads \
    --enable-__cxa_atexit \
    --disable-libmudflap \
    --with-gnu-ld --with-gnu-as \
    --disable-libssp --disable-libgomp \
    --disable-nls --disable-shared

make all-gcc install-gcc
make all-target-libgcc install-target-libgcc

ln -s $INSTALLDIR/lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc.a \
    $INSTALLDIR/lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc_sh.a
```

Table 6-5 explains the purpose of each of the command-line parameters passed to `configure`. Most of these parameters disable some feature that's not necessary to do what this compiler is meant to do: build the standard libraries. A few parameters tell the software what type of machine is being targeted so the proper code is generated.

***Table 6-5.*** *Bootstrap GCC Configuration Parameters*

| Configure Parameter | Meaning |
|---|---|
| `--build=$BUILDMACH` | The type of build system. The `configure` script can guess this, but it's a good habit to tell it what system is being used. |
| `--target=$TARGETMACH` | What machine is being targeted for the build. |
| `--prefix=$INSTALLDIR` | The installation directory of binutils. This is called `prefix` because this value is tacked on to the front of the normal installation directory. |
| `--enable-languages="c"` | Only the C language is necessary at this point. |
| `--disable-threads` | Disables thread support. Thread support comes from the libc library that this bootstrap compiler is building. |

| | |
|---|---|
| `--enable-__cxa_atexit` | Another C++-related configuration command describing how to implement the `atexit()` function call. |
| `--disable-libmudflap` | Library for runtime bounds checking. It isn't necessary. |
| `--disable-libssp` | Supplies protection from buffer overflow or stack-smashing attacks. |
| `--disable-libgomp` | Disables the inclusion of the OpenMP programming API for parallel programming. A parallel execution framework is unnecessary to build the full C library, and it also depends on the thread library, which isn't built at this stage. |
| `--disable-nls` | Removes internationalization. |
| `--disable-shared` | Specifies that the compiler doesn't have support for shared libraries. Shared library support is something else that is implemented with the yet-to-be-built C library. |
| `--enable-arch=armv4` | Gives the compilation process a hint about the target processor, so it can generate the right code. |
| `--with-gnu-as, --with-gnu-ld` | Instructs the configure script to use the GNU assembler and linker, as opposed to some other linker or assembler on the system. |

The make targets `all-gcc install-gcc` create the base compiler and install it in the prefix directory. The next parameters `all-target-gcc install-target-gcc` build a library that GCC uses during code generation, such as exception handling and floating-point emulation routines. This needs to be built before the Standard C Library, because this build relies on libgcc.

Last, there is a bit of trickery, when a symlink is created. The toolchain wants to link to the `libgcc_sh` file, which is for shared libraries, even when doing static linking. If your toolchain build is targeting a different platform or uses different software versions, you need to adjust this command appropriately.

## Creating Glibc Headers

The glibc headers are the used by the target system's compiler to get the definitions for the functions created by glibc. The GNU C Library, like others, isn't completely written in C; some of the code is written in the C preprocessor. There are also some circular dependencies: in order for glibc to compile, it needs some of the headers it generates that describe the host system.

Creating the headers involves configuring glibc and then running a portion of the build. You configure glibc now and then use the same configuration in the next step; you use the environment variables `CC`, `LD`, and `AS` to use the bootstrap cross-compiler, because you want to build code for the target machine:

```
[ -d $BUILDDIR/libc ] && rm -rf $BUILDDIR/libc
    mkdir -p $BUILDDIR/libc
    cd $BUILDDIR/libc
```

```
    echo "libc_cv_forced_unwind=yes" > config.cache
    echo "libc_cv_c_cleanup=yes" >> config.cache
    export PATH=$INSTALLDIR/bin:$PATH

    export CROSS=arm-none-linux-gnueabi
    export CC=${CROSS}-gcc
    export LD=${CROSS}-ld
    export AS=${CROSS}-as

    $LIBC_SRC/configure \
        --build=$BUILDMACH \
        --host=$TARGETMACH \
        --prefix=$SYSROOTDIR/usr \
        --with-headers=$SYSROOTDIR/usr/include \
        --config-cache \
        --enable-add-ons=glibc-ports-2.9,nptl \
        --enable-kernel=2.6.0

    make -k install-headers \
        cross_compiling=yes \
        install_root=$SYSROOTDIR

    ln -s $INSTALLDIR/lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc.a \
        $INSTALLDIR/lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc_eh.a
```

This is more convoluted than the others so far. First, the file `config.cache` is created with the following lines:

```
libc_cv_forced_unwind=yes
libc_cv_c_cleanup=yes
```

These lines prevent the configuration process from trying to run code compiled with the cross-compiler on the host system. Because the compiler generates code that runs on an ARM computer (and the compiler doesn't have the library support to create executables on Linux, which is why you're building glibc in the first place!) it won't run on the host system. By putting the results of these configuration parameters in a file and telling the configure script to use the cache, the code isn't executed, and the cached values is used instead. Table 6-6 lists the configuration values for glibc.

***Table 6-6.*** *Configuration Values for glibc*

| Configure Parameter | Meaning |
|---|---|
| `--build =$BUILDMACH` | The type of build system. The `configure` script can guess this, but it's a good habit to tell it what system is being used. Unless your computer has a big red switch on the side, this is the correct identifier. |

| | |
|---|---|
| `--target=$TARGETDIR` | What machine is being targeted for the build. |
| `--with-sysroot=$SYSROOTDIR/usr/include` | The root directory of the newly built system. When the toolchain is being built for an embedded system, this contains the files that are installed relative to the `/` directory. |
| `--enable-add-ons` | Tells the configuration script to see the `ports` directory (remember, your processor's support is in that directory) and all other glibc add-ons. The one add-on you're concerned with is nptl (native POSIX thread library); the rest are harmless at this point. |
| `--with-tls --with-__thread` | Options that work together to support the nptl feature by enabling thread local storage. |
| `--cache-file=config.cache` | Instructs `configure` to read the information in `config.cache`. Doing this results in your configure process skipping some problematic tests, because `configure` checks its cache before running code in order to execute faster. |
| `--with-headers=/opt/arm/include` | The location of the system headers to use with the build. These headers are used by glibc because many glibc functions rely on kernel system calls; these header files contain the necessary code to enable those calls. In addition, the headers include architecture-specific information used by glibc. |
| `--enable-kernel=2.6.0` | Instructs glibc that this software is running on a kernel that's 2.6 or later. |

The symlink that is created is due to the bootstrap toolchain attempting to link to the `libgcc_eh` file in all cases. This file contains routines for executables that use shared object and shouldn't be included, because you've built a static bootstrap compiler. The `libgcc.a` file has all the symbols you need; creating this symlink is a maneuver to satisfy the linker, because it insists that it can link to `libgcc_eh.a` even though no symbols are used from this file.

## Building Glibc

At this point, the directory `$INSTALLDIR` contains a toolchain that's complete enough to build the GNU C Standard Library. As with the other tools, this step involves a configuration and an installation step. But it has some differences in that the compiler used is the newly created cross-compiler, not the host compiler as in the prior steps:

```
[ -d $BUILDDIR/$LIBC ] && rm -rf $BUILDDIR/$LIBC
mkdir -p $BUILDDIR/$LIBC
cd $BUILDDIR/$LIBC
```

```
echo "libc_cv_forced_unwind=yes" > config.cache
echo "libc_cv_c_cleanup=yes" >> config.cache
export PATH=$INSTALLDIR/bin:$PATH

export CROSS=arm-none-linux-gnueabi
export CC=${CROSS}-gcc
export LD=${CROSS}-ld
export AS=${CROSS}-as

$SRCDIR/$LIBC/configure \
   --build=$BUILDMACH \
   --host=$TARGETMACH \
    --prefix=/usr \
    --with-headers=$SYSROOTDIR/usr/include \
   --config-cache \
   --enable-add-ons=glibc-ports-2.9,nptl \
   --enable-kernel=2.6.0

make -k install-headers cross_compiling=yes install_root=$SYSROOTDIR
ln -s /opt/arm/lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc.a \
   /opt/arm/lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc_eh.a

make
make install_root=${SYSROOT} install
```

The build for the glibc can take an hour or two on a reasonably fast computer. After the build completes and the software is installed, it's time for the final build of GCC. This produces a compiler that includes a complete Standard Library and is ready for use to build the kernel or an arbitrary user program.

## Building the Next GCC

With glibc compiled, you can now build a cross-compiler that has enough code to build an executable for the target; you need also build the GMP and MPFR libraries. These libraries use some of glibc and expect more than a bootstrap compiler installed on a system. This step builds a cross-compiler with all the dependencies necessary for these libraries. In addition, this cross-compiler has everything necessary for building static applications, so you can stop after this step if you're just building static libraries:

```
$ [ -d $BUILDDIR/final-gcc ] && rm -rf $BUILDDIR/final-gcc
$ mkdir -p $BUILDDIR/final-gcc
$ cd $BUILDDIR/final-gcc

$ echo "libc_cv_forced_unwind=yes" > config.cache
$ echo "libc_cv_c_cleanup=yes" >> config.cache
$ BUILD_CC=gcc
```

```
$ GCC_SRC/configure \
        --build=$BUILDMACH \
        --target=$TARGETMACH \
        --prefix=$INSTALLDIR \
        --with-sysroot=$SYSROOTDIR \
        --enable-languages=c \
        --with-gnu-as \
        --with-gnu-ld \
        --disable-multilib \
        --with-float=soft \
        --disable-sjlj-exceptions \
        --disable-nls --enable-threads=posix \
        --enable-long-longx

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

Table 6-7 lists the GCC build configuration options.

***Table 6-7.*** *GCC Build Configuration Options.*

| Configure Parameter | Meaning |
|---|---|
| `--enable-languages=c` | This cross-compiler is built with C support for now. |
| `-disable-sjlj-exceptions` | This is specific to C++ exception handling and means "set jump, long jump." GCC's C++ uses a table-driven exception-handling system that's more reliable. |
| `–enable-__cxa_atexit` | This is another C++-related configuration command describing how to implement the `atexit()` function call. |
| `-enable-threads=posix` | This instructs the glibc to include the POSIX threading API. The other option for this (aside from no thread support), LinuxThreads, is considered obsolete. |

## Building GMP and MPFR

The GMP library performs arithmetic on integers, floating-point, and irrational numbers. When the language needs to add multibyte numbers (and there's no opcode in the processor to handle such a task), GMP does the work. MPFR is a library that handles floating-point operations with high precision, greater than that of the processor.

### GMP

This library must be built first, because MPFR depends on it to build. This library builds much like the others. Notice how the `CC`, `LD`, and `AS` environment variables are set, because this code is cross-compiled

for the target. There is one additional trick—because the cross-compiler doesn't have shared library support, you need to pass in the `-static` flag as an argument to the compiler and request a static GMP library:

```
$ [ -d $BUILDDIR/gmp ] && rm -rf $BUILDDIR/gmp
$ mkdir -p $BUILDDIR/gmp
$ cd $BUILDDIR/gmp

$ export PATH=$INSTALLDIR/bin:$PATH

$ export CROSS=arm-none-linux-gnueabi
$ export CC=${CROSS}-gcc
$ export LD=${CROSS}-ld
$ export AS=${CROSS}-as
$ export CFLAGS=-static

$ $SRCDIR/$GMP/configure \
        --build=$BUILDMACH \
        --host=$TARGETMACH \
        --prefix=$INSTALLDIR \
        --disable-shared
$ make
$ make install
```

This code builds quickly. The parameters passed in to `configure` have been covered already in this chapter.

### MPFR

This toolkit for floating-point arithmetic is built next. The parameters for building it are as familiar as those for GMP, with the difference being that you must pass in the location of the GMP library:

```
$ [ -d $BUILDDIR/mpfr ] && rm -rf $BUILDDIR/mpfr
$ mkdir -p $BUILDDIR/mpfr
$ cd $BUILDDIR/mpfr

$ export PATH=$INSTALLDIR/bin:$PATH
$ export CROSS=arm-none-linux-gnueabi
$ export CC=${CROSS}-gcc
$ export LD=${CROSS}-ld
$ export AS=${CROSS}-as
$ export CFLAGS=-static

$ $SRCDIR/$MPFR/configure \
        --build=$BUILDMACH \
        --host=$TARGETMACH \
        --prefix=$INSTALLDIR \
        --with-gmp=$INSTALLDIR
```

```
$ make
$ make install
```

Like GMP, MPFR builds quickly, this process shouldn't take more than a few minutes.

## Building the Final GCC

Now that all the necessary libraries are in order, you can build the final toolchain. This is the final step in the process; after it's complete, the `$INSTALLDIR` directory contains a complete toolchain you can use for building applications for the target.

Configuring this toolchain is much like configuring the bootstrap toolchain. Notice that the `configure` statement points to the newly built GMP and MPFR libraries during the configure step and that the `CC`, `LD`, and `AS` variables aren't overridden because you want to use the compiler on your build system. The executable that is produced runs on the build machine, but you're producing code for the host machine:

```
$ [ -d $BUILDDIR/final-gcc-2 ] && rm -rf $BUILDDIR/final-gcc-2
$ mkdir -p $BUILDDIR/final-gcc-2
$ cd $BUILDDIR/final-gcc-2

$ echo "libc_cv_forced_unwind=yes" > config.cache
$ echo "libc_cv_c_cleanup=yes" >> config.cache

$  $GCC_SRC/configure \
        --build=$BUILDMACH \
        --target=$TARGETMACH \
        --prefix=$INSTALLDIR \
        --with-sysroot=$SYSROOTDIR \
        --enable-languages=c \
        --with-gnu-as \
        --with-gnu-ld \
        --disable-multilib \
        --with-float=soft \
        --disable-sjlj-exceptions \
        --disable-nls --enable-threads=posix \
        --disable-libmudflap \
        --disable-libssp \
        --enable-long-longx \
        --with-shared \
        --with-gmp=$INSTALLDIR \
        --with-mpfr=$INSTALLDIR

$  make
$  make install
```

This build runs for about an hour, even on a reasonably fast machine. When it's complete, your computer has a cross-compiler installed at `$INSTALLDIR`.

# Building Toolchains with Crosstool-NG

Creating a toolchain from scratch, as you can see, is tricky. The architecture and processor used in the example were chosen because of their widespread support in the Linux community. Some processors aren't as well supported and require patches or other non-obvious changes in order for the tools to build. The cross-compilation process, although completely open, is complex to the point that understanding it well enough to diagnose problems and keep abreast of the changes to the tools requires a serious time commitment.

A tool like crosstool-NG exists to encapsulate the knowledge about building toolchains and keeping the process up to date as the software advances. Crosstool-NG (the *NG* stands for "next generation," a nod to a television show of which you may have heard) grew from the efforts of Dan Kegel, who created a collection of scripts (the original crosstool) to build cross-compilers by changing some configuration files.[3] This script was later improved and morphed into the crosstool-NG project.

Using crosstool-NG reduces the effort for building a toolchain to a matter of choosing items from a menu. To get the tool, visit this URL for the latest version: `http://ymorin.is-a-geek.org/dokuwiki/projects/crosstool`. The following example uses version 13.2. By the time this book is published, the newest version will certainly be newer, given the amount activity on the project and the frequency of changes to the projects used by crosstool-NG.

Here are the steps necessary to create a toolchain using crosstool-NG. First, download the tools and extract them:

```
$ wget http://ymorin.is-a-geek.org/download/crosstool-ng/crosstool-ng-1.3.2.tar.bz2
$ tar xjf crosstool-ng-1.3.2.tar.bz2
```

Crosstool-NG has a short configure and installation process:

```
$./configure -prefix=~/ct
```

During the configuration process, crosstool-NG checks to make sure all the necessary utilities are installed, producing an error message if some are missing:

```
$make install
```

You can configure crosstool-NG to run from the directory where it was unpacked using the `-local` option when configuring. However, having the software installed in a separate directory is good practice. It includes samples that contain the typical tool settings for a processor and architecture. The samples in the distribution have been checked and should yield a toolchain without error. To list the samples, use the following command:

```
~/ct/bin/ct-ng list-samples
(clipped)
armeb-unknown-linux-uclibc
armeb-unknown-linux-uclibcgnueabi
arm-unknown-eabi
arm-unknown-elf
arm-unknown-linux-gnu
arm-unknown-linux-gnueabi
(clipped)
```

The toolchain you created earlier in this chapter, `arm-unknown-linux-gnueabi`, can also be created with this tool, which is what you do in the following few steps. To get additional information about any

---

[3] As somebody whose first job in embedded Linux was building toolchains, I was amazed how well this tool worked.

one of the toolchains listed, prefix any of them with `show-`. Here's the command for `arm-unknown-linux-gnueabi`:

```
$ ./bin/ct-ng show-arm-unknown-linux-gnueabi
    arm-unknown-linux-gnueabi  [g  ]
    OS        : linux-2.6.26.8
    GMP/MPFR  : gmp-4.2.4 / mpfr-2.3.2
    binutils  : binutils-2.19
    C compiler: gcc-4.3.2 (C,C++,Fortran,Java)
    C library : glibc-2.7
    Tools     :
```

You need to make a few additional changes in the configuration program, such as removing `sstrip` (a tool for reducing a program's size by a few bytes, which causes frequent compilation problems) and turning off the Fortran and Java compiler front ends, because these frequently fail to compile the intended target. You access the configuration program for crosstool-NG by doing the following. The configuration program interface should be familiar at this point:

```
$./bin/ct-ng menuconfig
```

To configure the build process using the `arm-unknown-linux-gnueabi` sample, do the following:

```
$./bin/ct-ng
```

The software does some work and outputs a message indicating that the build is ready. This process takes an hour or so. When it's complete, the toolchain is installed at `~/x-tools/arm-unknown-linux-gnueabi/bin` and is ready for use. The build log, stored in the `~/x-tools` directory as `build.log.bz2`, is verbose. Every file that's created, unpacked, or copied, along with each command executed, is kept in the log.

Considering that crosstool-NG builds completely from source and is largely implemented as bash scripts, using it to generate a toolchain is a better alternative than maintaining the code necessary to build a toolchain yourself. As the open source community updates and improves GCC, glibc, and its family of tools, the code around building this software also remains in flux, creating a maintenance task that crosstool-NG completes nicely.

# Creating the Root File System

Every Linux system has a root file system. On a desktop machine, the root file system is stored on a magnetic disk; but on an embedded system, the root file system may be stored on a flash device, packaged with the kernel, or, for some larger embedded systems, stored on a disk drive. During the bootup process, the kernel always looks for an initial RAM disk. The initial RAM disk comes from a file attached to the kernel during the build process that is uncompressed into RAM memory when the kernel starts. If a file to boot the system (initrd) is present, the kernel attempts to use the initial RAM disk.

Because the initial RAM disk is built during the kernel build process, it's being built before the kernel. When the root file system is completed, the kernel build process points at the root file system created during this step.

Although a fully functional root file system can be just one executable file or a few device files, for this example the root file system consists of a complete set of command-line tools typically found on a desktop system—compliments of the BusyBox project. BusyBox contains minimal implementations of the commands found on a desktop system, and all the utilities in BusyBox are compiled into one

executable and the file system. The file system is then populated with symlinks to the BusyBox executable. In the `main()` of BusyBox, a switch uses the name of the file (which is the name of the symlink used to invoke the program) to figure out what code to execute.

## Configuring the Environment

Before you build the root file system, create an installation folder for the executables, libraries, and other files that go into the root file system. From the perspective of the machine that eventually uses the root file system, the contents below this directory are the / of that machine. This is referred to as `$RFS` in the remainder of this section, because the directory can be in any arbitrary location:

```
$ export RFS=$RFS
$ mkdir –p $RFS
```

## Building and Installing BusyBox

Start by downloading and unpacking the source code

```
$ http://busybox.net/downloads/busybox-1.13.3.tar.bz2
$tar xjf http://busybox.net/downloads/busybox-1.13.3.tar.bz2
```

The method for configuring the tool should now be familiar. For example purposes, use the `defconfig` option that enables just about every BusyBox option except debugging:

```
$ make defconfig
$ make menuconfig
```

In the `menuconfig` interface, do these two things:

1. Set the Cross-Compiler Prefix under the build options to point to the newly created toolchain. This option is under BusyBox Settings" ➤ Build Options Menu. Set this value to the entire path and name of the cross-compiler. For example, using the cross-compiler built in this chapter, the value is `/opt/arm/bin/arm-none-linux-gnueabi-`. The trailing - is important!
2. Change the BusyBox installation to $RFS. Look for this option under BusyBox Settings ➤ Installation Options. This is where BusyBox will put its executable and symlinks after the build process.

After you've made these changes, the next step is starting a build:

```
$ make
$ make install
```

The folder $RFS contains the start of a root file system:

```
bin
linuxrc -> bin/busybox
sbin
usr
```

This root file system isn't yet complete. It's missing some critical components such as the libraries, device files, and a few mount points. The next sections cover how to gather the rest of the components from the toolchain.

## Libraries

The BusyBox project was built with shared libraries—the glibc shared libraries in this case. These libraries need to be in the root file system when the board runs; otherwise the program will refuse to load. In addition to the shared libraries, the system needs the program that loads the program and performs the dynamic linking of the program and the shared libraries, usually called `ld.so` or `ld-linux.so`. Just like everything else in Linux, this program doesn't have a fixed name—it can have any arbitrary name, so long as all parties agree on the label.

On a desktop Linux system, to find the name of the library loader, use the `ldd` command to show the libraries loaded by a program:

```
$ ldd `which ls`
        linux-gate.so.1 =>  (0xb7f6c000)
        librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7f4f000)
        libselinux.so.1 => /lib/libselinux.so.1 (0xb7f36000)
        libacl.so.1 => /lib/libacl.so.1 (0xb7f2f000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7de0000)
        libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7dc8000)
        /lib/ld-linux.so.2 (0xb7f6d000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7dc4000)
        libattr.so.1 => /lib/libattr.so.1 (0xb7dbf000)
```

Look at the list, and find the program that starts with an absolute path. In this system, the file is `/lib/ld-linux.so.2`. When running a program linked with shared libraries, the operating system runs `/lib/ld-linux.so.2` and passes the name of the program to run (along with its parameters) to the loader.

To generate this output, `ldd` runs the program in question. This can't be done with the cross-compiled BusyBox, because it contains binary code for an ARM processor, and the build was likely done on an Intel PC. The alternative to using `ldd` is the much cruder but effective solution of looking for strings in the executable that start with `lib`:

```
$strings $RFS/bin/busybox | grep lib
/lib/ld-linux.so.3
libm.so.6
libc.so.6
__libc_start_main
/lib/modules
/var/lib/misc/udhcpd.leases
/lib/firmware
/var/lib/hwclock/adjtime
```

The files you're interested are `/lib/ld-linux.so.3` (the loader), `libm.so.6`, and `libc.so.6`. These files are symlinks stored under the `$SYSROOT` directory using during the toolchain build. To get them in the root file system, create the directory and copy the files:

```
$ cd $RFS
$ Mkdir lib
$ pushd $SYSROOT/usr/lib
$ cp libc.so.6 libm.so.6 ld-linux.so.3 /home/gene/build-gcc/rfs/lib/
$ popd
```

To avoid the problem of shared libraries, BusyBox can be linked directly to the libc libraries (static linking) so that the `.so` files aren't needed on the target. As an experiment, try static linking: use the search in BusyBox's `menuconfig` to find the menu item, rebuild the root file system, and see how the size changes.

## Creating Device Nodes and Directories

Linux requires that two devices be present in order to work correctly: console and null. A device node is a way for a user program to communicate with a kernel device driver. These are created on the file system so the kernel build can gather them into the RAM disk it uses to boot the file system:

```
$cd $RFS
$mkdir dev
$cd dev
$sudo mknod console c 5 1
$sudo mknod null c 3 1
```

This is one of the few commands during this process executed as root. Be careful and make sure the commands are executed in the right directory. Later, the book discusses a way to do this without being root, when more time is dedicated to the kernel build process.

A root file system also needs a few extra directories: `root`, `tmp`, and `proc`. The `proc` directory isn't strictly necessary, but many programs rely on the proc file system being mounted at `/proc` to operate correctly:

```
$cd $RFS
$mkdir tmp root etc proc
```

The system will start without these directories, but having them present prevents some error messages from appearing when the kernel loads. If you're fretting over every byte in the root file system, you can leave these out to save a little space (even empty directories consume a few bytes).

## Finishing Touches

The system requires a few additional files that are read by user programs: the user and group files and the inittab. You can create these files from the command line:

```
$ echo "root::0:0:root:/root:/bin/sh" > $RFS/etc/passwd
$ echo "root:x:0:" > $RFS/etc/groups
$ echo "::respawn:/sbin/getty -L ttyAMA0 115200 xterm > $RFS/etc/inittab
```

The first two lines create a group and user for the root user. The third line creates an inittab that starts a terminal login when the system starts. When a Linux system boots, it reads the inittab file to figure out what to run. In this case, the inittab instructs the system to run `/sbin/getty` at the start and to restart the program when it stops.

In a deployed Linux system, this file contains references to additional scripts that start programs necessary for the system, such as an HTTP server, or perform housekeeping tasks like configuring network adapters. The most important aspect of the inittab is that one program is configured for respawn: when init no longer has anything to run, it stops, and so does the kernel.

This system is built using an initial RAM disk. This is part of the kernel and is loaded into memory during boot time. The kernel attempts to run the file `/linuxrc`; if this file isn't present or can't be loaded, the kernel continues the standard booting process by mounting a root file system and as specified on the kernel command line. This is covered in the next section.

## Building the Kernel

Now that the toolchain and root file system builds have been completed, the next step is to compile the kernel for the target machine. By now, the kernel configuration program should be familiar, because several of the projects in this chapter have adopted it as their configuration interface. In addition, the process of configuring a project so that the source tree is in the correct state before building should also be familiar. Early in the chapter, to get the header files necessary to build the toolchain, you configured the kernel for an ARM processor running on one of ARM's development boards using this command executed from the top-level kernel directory:

```
$ make ARCH=arm integrator_defconfig
```

This command results in the kernel finding the file `integrator_defconfig` under the ARM architecture directory and copying those settings into the current configuration, stored at `.config`.[4] This file was contributed by ARM to make it easier to build a Linux kernel for this board and isn't the only `defconfig` present. To see a list of all the `defconfig` files, do the following:

```
$make ARCH=arm help
```

This lists the make targets and the `_defconfig` files for the architecture. ARM has by far the largest count of `defconfig` files. This command doesn't have any side effects, so feel free to view the support for other architectures by passing in other values for `ARCH`. You can find the population of acceptable values for `ARCH` by looking at the first level of subdirectories in the kernel sources under the `arch` directory. For example:

```
$find ./arch -type d -maxdepth 1
```

Because the kernel has already been configured to build for the target device, only a little extra configuration is necessary: updating the kernel command line to boot up using some parameters specific to this build of the kernel and root file system. The kernel command line is like the command line on a utility run from a shell: the parameters get passed in to the kernel at bootup time and let you change how the kernel works without recompilation. In this case, make the kernel command line this value:

```
console=ttyAMA0
```

---

[4] The file `.config` is a hidden file. To see it in a directory listing, use the `-a` argument to `ls`, which shows all files in the directory.

You set this value through the kernel configuration tool by running the following:

```
$ make ARCH=arm menuconfig
```

The following appears before the configuration menu, during compilation of the software that draws the configuration menu. This happens the first time the kernel menu configuration programs runs:

```
(clipped)
  HOSTCC  scripts/kconfig/lxdialog/inputbox.o
  HOSTCC  scripts/kconfig/lxdialog/menubox.o
  HOSTCC  scripts/kconfig/lxdialog/textbox.o
  HOSTCC  scripts/kconfig/lxdialog/util.o
  HOSTCC  scripts/kconfig/lxdialog/yesno.o
(clipped)
```

Look for Boot Options in the Linux kernel configuration top level screen, as shown in Figure 6-2

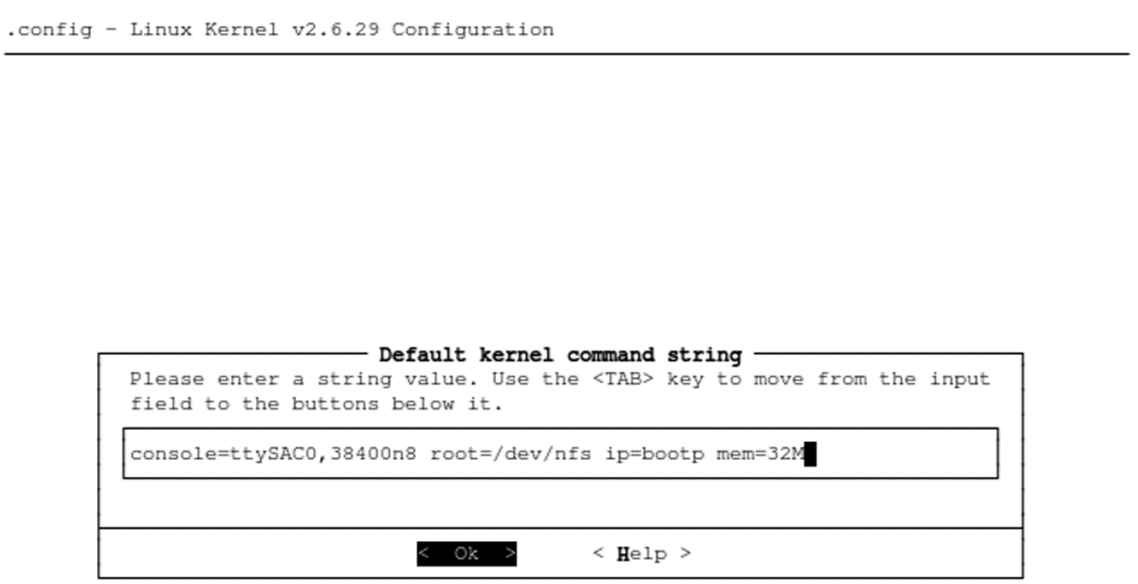


***Figure 6-2.*** *The Linux Kernel Configuration top level menu*

The settings for the kernel command line are at the following menu location: Boot Options ➤ Default Kernel Command String. To change the settings, highlight the item in the menu, and press Enter. The window shown in Figure 6-3 appears on screen.

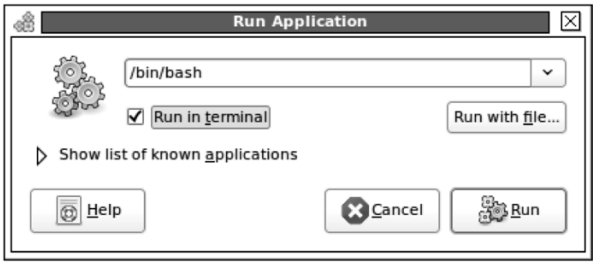


***Figure 6-3.*** *Setting the value of the kernel command line*

The user interface for entering strings in the kernel configuration program is frustrating. You can't use the cursor keys to navigate to some character and begin typing. Instead, you must make changes by backspacing and replacing text. The easiest way to make changes is to compose them in an editor and paste them into this control. After you update the boot parameters, select OK and Exit on the subsequent menus. The software asks to save changes, which you should affirm. The kernel is now ready to build.

When you run make from the command line for the kernel, you need to pass in the cross-compiler prefix and the architecture:

```
$ export PATH=$PATH:/opt/arm-tools/bin
$ ake zImage ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

This command line requests that a zImage be built and prefixes the compiler name (GCC) with `CROSS_COMPILE`. How do you know the right target—not xImage or ouiejsdf, for example? Table 6-8 lists the common build targets for Linux.

***Table 6-8.*** *Commond Linux Target Build Options*

| Target | Explanation |
|---|---|
| `zImage` | This is a zlib compressed image of the kernel, ready to boot on a board. |
| `uImage` | This is a zlib compressed image, ready to boot on a board that's running the U-Boot boot loader. It's much like the zImage but contains additional data that makes booting with U-Boot more efficient. |
| `bzImage` | This is a big zImage—the type of image used with the boot loaders on x86 hosts. |
| `vmlinux` | The vmlinux target builds the Linux kernel executable image. This closely resembles what's in memory for the Linux kernel and is frequently used for debugging. |
| `srecImage` | Very few board use S-records. For those that do, this is the target. An S-record is a format where each line in the file starts with *S*. |
| `xipImage` | XIP stands for Execute in Place. This image is formatted so that it can be put into a flash partition and run from there, without getting loaded into RAM first. |

If you've built the kernel for an Intel x86 architecture, the target bzImage is familiar and is still the right value when you're building for x86 embedded boards. The target in question in an ARM board, which expects a zImage. The documentation for a board's boot loader specifies the format of binary it requires.

The kernel compilation takes about 30 minutes, give or take. The last few lines of the kernel build show what file was created:

```
Kernel: arch/arm/boot/Image is ready
Kernel: arch/arm/boot/zImage is ready
```

Congratulations! The kernel is completed, and the root file system has been built as part of kernel an initial RAM disk. When this kernel boots, it will start a command shell and wait for a login. The root file system has been built into the kernel, and the kernel has a copy of what was include in the `./usr` directory under the kernel root with the name `initramfs_data.cpio.gz`.

---

■ **Note** Don't have an ARM integrator handy? Yes, you do. The kernel and root file system that you've built can be booted using QEMU. Use the `qemu-system-arm` command with the `-kernel` and `-initrd` of the `initramfs_data.cpio.gz` created during the build.

---

# Troubleshooting Booting Problems

What if it didn't work? The software built in this chapter is remarkably resilient and is configured to do the right thing with minimal input. Still, after all this work, more often than not, you may have skipped a step, or some other problem may require debugging. In general, systems don't boot for one of the following reasons:

- The kernel isn't properly configured for the board.
- The root file system can't be mounted.
- The root file system `init` program doesn't run.
- It's booting, but the serial connection parameters are wrong.

The easiest way to diagnose board-booting problems is to have a running Linux for the board. Because nearly every board vendor includes a Linux distribution of one sort of another, this is a less unreasonable statement than in years past. When Linux is up and running, you can use it as a test-bed for the code that's not.

## Improperly Configured Board

This occurs when the following output appears, followed by nothing.

```
Uncompressing
Linux.....................................................................↵
..................... done, booting the kernel.
```

This can be confused with the kernel actually booting but the configuration parameters are incorrect. Double-check that the cables are connected and that the communication parameters are

correct. Then, check this item again. If the kernel is indeed stopping here, check the kernel configuration program to ensure that the board and processor are correct. Next, check that that the toolchain is correct; some users with several toolchains mistakenly use the wrong one when compiling the kernel.

If all seems well at this point, look at the distribution of Linux included with the board and diff the configuration file against the one used to build the kernel. If the configuration file isn't present, it's sometimes built into the kernel and resides at `/proc/config.gz`. The kernel includes a utility that can extract this information from a kernel file called `./scripts/extract-ikconfig`. This file writes the output to the console, so it's best to redirect the output to a file:

```
cd $ <kernel directory>
./scripts/extract-ikconfig <path to kernel zImage>  > ~/kernel-config
```

After it's extracted, you can use this file to configure the kernel by making a copy of it named `.config` in the top-level kernel directory. After making the copy, issue a `make oldconfig` command to have the build reread the configuration file and configure the environment:

```
$cp ~/kernel-config <kernel directory>/.config
$make ARCH=arm oldconfig
```

When the kernel builds again, it may ask for default values for configuration parameters added because the `oldconfig` was created. The best thing to do is accept the defaults, unless you know otherwise. When the kernel finishes building, it's configured close to how the booting kernel was built.

## The Root File System Can't Be Mounted

The kernel looks for an initial RAM disk root file system and then for the root file system as instructed on the command line. If the kernel can't use the initial RAM file system, it probably doesn't have the `linuxrc` file in the root directory or this file isn't executable. Checking for its existence isn't difficult, but making sure it can run is more of a trick. This is where having a running kernel and RFS comes in handy, because you can copy the files to the board and execute them to see if they run, or you can mount the file system via NFS and test over the network.

Most problems with the file system not being mounted come from not entering the correct directory for the root file system in the kernel configuration program. The kernel build fails silently if it can't find the directory or file containing the root file system.

## The Root File System init Program Doesn't Run

This problem is generally a result of not having the libraries and other supporting files in the root file system. If BusyBox was linked with shared libraries, make sure they're all present in the `/lib` directory and that the linker file (`ld-linux.3`) is present as well. This error is difficult to find because the kernel produces an error like the following if this is indeed the problem, and it doesn't seem to have anything to do with the `init` file not working:

```
Root-NFS: No NFS server available, giving up.
VFS: Unable to mount root fs via NFS, trying floppy.
VFS: Cannot open root device "<NULL>" or unknown-block(2,0)
Please append a correct "root=" boot option; here are the available partitions:
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0)
```

What really throws people new to Linux is the message about no NFS server being available. This appears even if there's no attempt to use an NFS server. When you're chasing down library dependency problems, using `ldd` on the running Linux host is the best way to go. On the build system, you can't run `ldd` because it tries to execute the program in question and its instruction set is probably different than the host system, resulting in an error. If the problem is still mysterious, copy all the `.so` files from the toolchain's `sysroot` directory into the `/libfolder` of the target, and begin eliminating files one at time. This is a crude but effective method.

# Distributing the Distribution

The Linux distribution created for a project needs to be bundled up and transmitted to people working on the project. Because most Linux developers work for companies with small teams, the most common practice is to create archives with the toolchain binaries and a snapshot of the root file system in binary form. The sources for these projects are then checked into source control along with the build scripts. For this project, you create the toolchain tar file like so:

```
$ tar cjf arm-tools.tar.bz2 $INSTALLDIR
```

This archive can then be unpacked on a developer's machine in any location and used to compile programs.

You can archive the root file system in the same manner:

```
$ tar cjf arm-tools.tar.bz2 $RFS
```

The user can then unpack this root file system and use it to boot the board or add it into a kernel build.

Getting the kernel ready for packing is a little more complex. The zImage file used to boot the system can be stored and given to people working on the project; however, those doing kernel work need to rebuild the kernel and should be do so with the same configuration used to create the distribution.

The kernel configuration process creates a `.config` file to hold the user's selections. You can copy this file into the appropriate `arch/configs` directory with the name `<`*something*`>_config`, where *something* is an arbitrary identifier. In the case of this project, the command looks like

```
$ cp .config arch/arm/configs/example_defconfig
```

After the configuration is saved, you should clean the kernel directory of all output using this command:

```
$ make ARCH=arm distclean
```

This command does a complete clean of the kernel tree, even deleting files that look like editor backup files. When this is finished, you can archive the kernel tree into a compressed tar file, ready for distribution. When users receive the file, they need to uncompress it and do the following to re-create the build environment:

```
make ARCH=arm exmaple_defconfig
```

Then they're ready to do their development and build the kernel.

## Wrapping Up

Creating a Linux distribution from source files isn't terribly difficult. It's a task that's complex by virtue of the number of steps you must follow and the amount of detail at every step. Perhaps the biggest problem is the large cycle time between experimentation and seeing results. Many times, a toolchain or kernel build requires 30 minutes of compilation to test just a small change, and that latency makes experimentation unwieldy.