2
Learning about Toolchains

The toolchain is the first element of embedded Linux and the starting point of your project. You will use it to compile all the code that will run on your device. The choices you make at this early stage will have a profound impact on the final outcome. Your toolchain should be capable of making effective use of your hardware by using the optimum instruction set for your processor. It should support the languages that you require and have a solid implementation of the **Portable Operating System Interface** (**POSIX**) and other system interfaces.

Your toolchain should remain constant throughout the project. In other words, once you have chosen your toolchain, it is important to stick with it. Changing compilers and development libraries in an inconsistent way during a project will lead to subtle bugs. That being said, it is still best to update your toolchain when security flaws or bugs are found.

Obtaining a toolchain can be as simple as downloading and installing a TAR file, or it can be as complex as building the whole thing from source code. In this chapter, I take the latter approach, with the help of a tool called **crosstool-NG**, so that I can show you the details of creating a toolchain. Later on, in *Chapter 6*, *Selecting a Build System*, I will switch to using the toolchain generated by the build system, which is the more usual means of obtaining a toolchain. When we get to *Chapter 14*, *Starting with BusyBox runit*, we'll save ourselves some time by downloading a prebuilt Linaro toolchain to use with Buildroot.

In this chapter, we will cover the following topics:

- Introducing toolchains
- Finding a toolchain
- Building a toolchain using the crosstool-NG tool
- The anatomy of a toolchain
- Linking with libraries – static and dynamic linking
- The art of cross-compiling

Technical requirements

To follow along with the examples, make sure you have the following:

- A Linux-based host system with `autoconf`, `automake`, `bison`, `bzip2`, `cmake`, `flex`, `g++`, `gawk`, `gcc`, `gettext`, `git`, `gperf`, `help2man`, `libncurses5-dev`, `libstdc++6`, `libtool`, `libtool-bin`, `make`, `patch`, `python3-dev`, `rsync`, `texinfo`, `unzip`, `wget`, and `xz-utils` or their equivalents installed.

I recommend using Ubuntu 20.04 LTS or later since the exercises in this chapter were all tested against that Linux distribution at the time of writing. Here is the command to install all the required packages on Ubuntu 20.04 LTS:

```
$ sudo apt-get install autoconf automake bison bzip2 cmake \
flex g++ gawk gcc
gettext git gperf help2man libncurses5-dev libstdc++6 libtool \
libtool-bin make
patch python3-dev rsync texinfo unzip wget xz-utils
```

All of the code for this chapter can be found in the `Chapter02` folder from the book's GitHub repository: `https://github.com/PacktPublishing/Mastering-Embedded-Linux-Programming-Third-Edition`

Introducing toolchains

A toolchain is a set of tools that compiles source code into executables that can run on your target device and includes a compiler, a linker, and runtime libraries. Initially, you need one to build the other three elements of an embedded Linux system: the bootloader, the kernel, and the root filesystem. It has to be able to compile code written in assembly, C, and C++ since these are the languages used in the base open source packages.

Usually, toolchains for Linux are based on components from the GNU project (`http://www.gnu.org`), and that is still true in the majority of cases at the time of writing. However, over the past few years, the **Clang** compiler and the associated **Low Level Virtual Machine** (**LLVM**) project (`http://llvm.org`) have progressed to the point that it is now a viable alternative to a GNU toolchain. One major distinction between LLVM and GNU-based toolchains is the licensing; LLVM has a BSD license while GNU has the GPL.

There are some technical advantages to Clang as well, such as faster compilation and better diagnostics, but GNU GCC has the advantage of compatibility with the existing code base and support for a wide range of architectures and operating systems. While it took some years to get there, Clang can now compile all the components needed for embedded Linux and is a viable alternative to GNU. To learn more about that, see `https://www.kernel.org/doc/html/latest/kbuild/llvm.html`.

There is a good description of how to use Clang for cross-compilation at `https://clang.llvm.org/docs/CrossCompilation.html`. If you would like to use it as part of an embedded Linux build system, the EmbToolkit (`https://embtoolkit.org`) fully supports both GNU and LLVM/Clang toolchains, and various people are working on using Clang with Buildroot and the Yocto Project. I will cover embedded build systems in *Chapter 6*, *Selecting a Build System*. Meanwhile, this chapter focuses on the GNU toolchain as it is still the most popular and mature toolchain for Linux.

A standard GNU toolchain consists of three main components:

- **Binutils**: A set of binary utilities including the assembler and the linker. It is available at `http://gnu.org/software/binutils`.
- **GNU Compiler Collection** (**GCC**): These are the compilers for C and other languages, which, depending on the version of GCC, include C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go. They all use a common backend that produces assembler code, which is fed to the GNU assembler. It is available at `http://gcc.gnu.org/`.
- **C library**: A standardized **application program interface** (**API**) based on the POSIX specification, which is the main interface to the operating system kernel for applications. There are several C libraries to consider, as we shall see later on in this chapter.

Along with these, you will need a copy of the Linux kernel headers, which contain definitions and constants that are needed when accessing the kernel directly. Right now, you need them to be able to compile the C library, but you will also need them later when writing programs or compiling libraries that interact with particular Linux devices, for example, to display graphics via the Linux frame buffer driver. This is not simply a question of making a copy of the header files in the `include` directory of your kernel source code. Those headers are intended for use in the kernel only and contain definitions that will cause conflicts if used in their raw state to compile regular Linux applications.

Instead, you will need to generate a set of sanitized kernel headers, which I have illustrated in *Chapter 5*, *Building a Root Filesystem*.

It is not usually crucial whether the kernel headers are generated from the exact version of Linux you are going to be using or not. Since the kernel interfaces are always backward compatible, it is only necessary that the headers are from a kernel that is the same as, or older than, the one you are using on the target.

Most people would consider the **GNU Debugger** (**GDB**) to be part of the toolchain as well, and it is usual that it is built at this point. I will talk about GDB in *Chapter 19*, *Debugging with GDB*.

Now that we've talked about kernel headers and seen what the components of a toolchain are, let's look at the different types of toolchains.

Types of toolchains

For our purposes, there are two types of toolchain:

- **Native**: This toolchain runs on the same type of system (sometimes the same actual system) as the programs it generates. This is the usual case for desktops and servers, and it is becoming popular on certain classes of embedded devices. The Raspberry Pi running Debian for ARM, for example, has self-hosted native compilers.
- **Cross**: This toolchain runs on a different type of system than the target, allowing the development to be done on a fast desktop PC and then loaded onto the embedded target for testing.

Almost all embedded Linux development is done using a cross-development toolchain, partly because most embedded devices are not well suited to program development since they lack computing power, memory, and storage, but also because it keeps the host and target environments separate. The latter point is especially important when the host and the target are using the same architecture, `x86_64`, for example. In this case, it is tempting to compile natively on the host and simply copy the binaries to the target.

This works up to a point, but it is likely that the host distribution will receive updates more often than the target, or that different engineers building code for the target will have slightly different versions of the host development libraries. Over time, the development and target systems will diverge, and you will violate the principle that the toolchain should remain constant throughout the life of the project. You can make this approach work if you ensure that the host and the target build environments are in lockstep with each other. However, a much better approach is to keep the host and the target separate, and a cross toolchain is the way to do that.

However, there is a counter argument in favor of native development. Cross development creates the burden of cross compiling all the libraries and tools that you need for your target. We will see later in the section titled *The art of cross compiling* that cross development is not always simple because many open source packages are not designed to be built in this way. Integrated build tools, including Buildroot and the Yocto Project, help by encapsulating the rules to cross-compile a range of packages that you need in typical embedded systems, but if you want to compile a large number of additional packages, then it is better to natively compile them. For example, building a Debian distribution for the Raspberry Pi or BeagleBone using a cross compiler would be very hard. Instead, they are natively compiled.

Creating a native build environment from scratch is not easy. You would still need a cross compiler at first to create the native build environment on the target, which you then use to build the packages. Then, in order to perform the native build in a reasonable amount of time, you would need a build farm of well-provisioned target boards, or you may be able to use **Quick EMUlator** (**QEMU**) to emulate the target.

Meanwhile, in this chapter, I will focus on a more mainstream cross compiler environment, which is relatively easy to set up and administer. We will start by looking at what distinguishes one target CPU architecture from another.

CPU architectures

The toolchain has to be built according to the capabilities of the target CPU, which includes the following:

- **CPU architecture**: **ARM**, **Microprocessor without Interlocked Pipelined Stages** (**MIPS**), x86_64, and so on.
- **Big- or little-endian operation**: Some CPUs can operate in both modes, but the machine code is different for each.

- **Floating point support**: Not all versions of embedded processors implement a hardware floating-point unit, in which case the toolchain has to be configured to call a software floating-point library instead.
- **Application Binary Interface** (**ABI**): The calling convention used for passing parameters between function calls.

With many architectures, the ABI is constant across the family of processors. One notable exception is ARM. The ARM architecture transitioned to the **Extended Application Binary Interface** (**EABI**) in the late 2000s, resulting in the previous ABI being named the **Old Application Binary Interface** (**OABI**). While the OABI is now obsolete, you'll continue to see references to EABI. Since then, the EABI has split into two, based on the way the floating-point parameters are passed.

The original EABI uses general-purpose (integer) registers, while the newer **Extended Application Binary Interface Hard-Float** (**EABIHF**) uses floating point registers. The EABIHF is significantly faster at floating-point operations, since it removes the need for copying between integer and floating-point registers, but it is not compatible with CPUs that do not have a floating-point unit. The choice, then, is between two incompatible ABIs; you cannot mix and match the two, and so you have to decide at this stage.

GNU uses a prefix to the name of each tool in the toolchain, which identifies the various combinations that can be generated. It consists of a tuple of three or four components separated by dashes, as described here:

- **CPU**: This is the CPU architecture, such as ARM, MIPS, or x86_64. If the CPU has both endian modes, they may be differentiated by adding `el` for little-endian or `eb` for big-endian. Good examples are little-endian MIPS, `mipsel`, and big-endian ARM, `armeb`.
- **Vendor**: This identifies the provider of the toolchain. Examples include `buildroot`, `poky`, or just `unknown`. Sometimes it is left out altogether.
- **Kernel**: For our purposes, it is always `linux`.
- **Operating system**: A name for the user space component, which might be `gnu` or `musl`. The ABI may be appended here as well, so for ARM toolchains, you may see `gnueabi`, `gnueabihf`, `musleabi`, or `musleabihf`.

You can find the tuple used when building the toolchain by using the `-dumpmachine` option of `gcc`. For example, you may see the following on the host computer:

```
$ gcc -dumpmachine
x86_64-linux-gnu
```

This tuple indicates a CPU of `x86_64`, a kernel of `linux`, and a user space of `gnu`.

> **Important note**
>
> When a native compiler is installed on a machine, it is normal to create links to each of the tools in the toolchain with no prefixes, so that you can call the C compiler with the `gcc` command.

Here is an example using a cross compiler:

```
$ mipsel-unknown-linux-gnu-gcc -dumpmachine
mipsel-unknown-linux-gnu
```

This tuple indicates a CPU of little-endian MIPS, an `unknown` vendor, a kernel of `linux`, and a user space of `gnu`.

Choosing the C library

The programming interface to the Unix operating system is defined in the C language, which is now defined by the POSIX standards. The **C library** is the implementation of that interface; it is the gateway to the kernel for Linux programs, as shown in the following diagram. Even if you are writing programs in another language, maybe Java or Python, the respective runtime support libraries will have to call the C library eventually, as shown here:

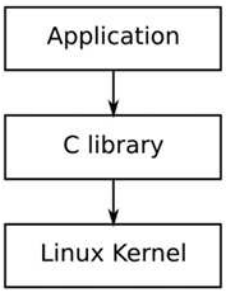

Figure 2.1 – C library

Whenever the C library needs the services of the kernel, it will use the kernel system call interface to transition between user space and kernel space. It is possible to bypass the C library by making the kernel system calls directly, but that is a lot of trouble and almost never necessary.

There are several C libraries to choose from. The main options are as follows:

- **glibc**: This is the standard GNU C library, available at `https://gnu.org/software/libc`. It is big and, until recently, not very configurable, but it is the most complete implementation of the POSIX API. The license is LGPL 2.1.
- **musl libc**: This is available at `https://musl.libc.org`. The `musl libc` library is comparatively new but has been gaining a lot of attention as a small and standards-compliant alternative to GNU `libc`. It is a good choice for systems with a limited amount of RAM and storage. It has an MIT license.
- **uClibc-ng**: This is available at `https://uclibc-ng.org`. `u` is really a Greek `mu` character, indicating that this is the microcontroller C library. It was first developed to work with uClinux (Linux for CPUs without memory management units) but has since been adapted to be used with full Linux. The `uClibc-ng` library is a fork of the original `uClibc` project (`https://uclibc.org`), which has unfortunately fallen into disrepair. Both are licensed with LGPL 2.1.
- **eglibc**: This is available at `http://www.eglibc.org/home`. Now obsolete, `eglibc` was a fork of `glibc` with changes to make it more suitable for embedded usage. Among other things, `eglibc` added configuration options and support for architectures not covered by `glibc`, in particular the PowerPC e500 CPU core. The code base from `eglibc` was merged back into `glibc` in version 2.20. The `eglibc` library is no longer maintained.

So, which to choose? My advice is to use `uClibc-ng` only if you are using uClinux. If you have a very limited amount of storage or RAM, then `musl libc` is a good choice, otherwise, use `glibc`, as shown in this flow chart:

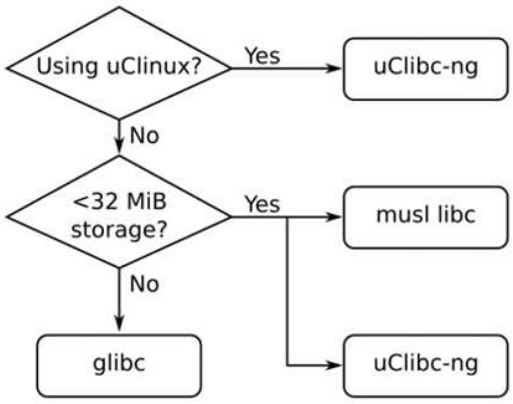

Figure 2.2 – Choosing a C library

Your choice of C library could limit your choice of toolchain since not all pre-built toolchains support all C libraries.

Finding a toolchain

You have three choices for your cross-development toolchain: you may find a ready-built toolchain that matches your needs; you can use one generated by an embedded build tool, which is covered in *Chapter 6*, *Selecting a Build System*; or you can create one yourself as described later in this chapter.

A pre-built cross toolchain is an attractive option, in that you only have to download and install it, but you are limited to the configuration of that particular toolchain and you are dependent on the person or organization you got it from.

Most likely, it will be one of these:

- An SoC or board vendor. Most vendors offer a Linux toolchain.
- A consortium dedicated to providing system-level support for a given architecture. For example, Linaro, (`https://www.linaro.org`) have pre-built toolchains for the ARM architecture.
- A third-party Linux tool vendor, such as Mentor Graphics, TimeSys, or MontaVista.
- The cross-tool packages for your desktop Linux distribution. For example, Debian-based distributions have packages for cross compiling for ARM, MIPS, and PowerPC targets.
- A binary SDK produced by one of the integrated embedded build tools. The Yocto Project has some examples at `http://downloads.yoctoproject.org/releases/yocto/yocto-[version]/toolchain`.
- A link from a forum that you can't find anymore.

In all of these cases, you have to decide whether the pre-built toolchain on offer meets your requirements. Does it use the C library you prefer? Will the provider give you updates for security fixes and bugs, bearing in mind my comments on support and updates from *Chapter 1*, *Starting Out*. If your answer is no to any of these, then you should consider creating your own.

Unfortunately, building a toolchain is no easy task. If you truly want to do the whole thing yourself, take a look at *Cross Linux From Scratch* (`https://trac.clfs.org`). There you will find step-by-step instructions on how to create each component.

A simpler alternative is to use crosstool-NG, which encapsulates the process into a set of scripts and has a menu-driven frontend. You still need a fair degree of knowledge, though, just to make the right choices.

It is simpler still to use a build system such as Buildroot or the Yocto Project, since they generate a toolchain as part of the build process. This is my preferred solution, as I have shown in *Chapter 6*, *Selecting a Build System*.

With the ascendance of crosstool-NG, building your own toolchain is certainly a valid and viable option. Let's look at how to do that next.

Building a toolchain using crosstool-NG

Some years ago, Dan Kegel wrote a set of scripts and makefiles for generating cross-development toolchains and called it crosstool (`http://kegel.com/crosstool/`). In 2007, Yann E. Morin used that base to create the next generation of crosstool, crosstool-NG (`https://crosstool-ng.github.io`). Today it is by far the most convenient way to create a standalone cross toolchain from source.

In this section, we will use crosstool-NG to build toolchains for the BeagleBone Black and QEMU.

Installing crosstool-NG

Before you can build crosstool-NG from source, you will first need to install a native toolchain and some build tools on your host machine. See the section on *Technical requirements* at the beginning of this chapter for crosstool-NG's complete list of build and runtime dependencies.

Next, get the current release from the crosstool-NG Git repository. In my examples, I have used version `1.24.0`. Extract it and create the frontend menu system, `ct-ng`, as shown in the following commands:

```
$ git clone https://github.com/crosstool-ng/crosstool-ng.git
$ cd crosstool-ng
$ git checkout crosstool-ng-1.24.0
$ ./bootstrap
$ ./configure --prefix=${PWD}
$ make
$ make install
```

The `--prefix=${PWD}` option means that the program will be installed into the current directory, which avoids the need for root permissions, as would be required if you were to install it in the default location `/usr/local/share`.

We now have a working installation of crosstool-NG that we can use to build cross toolchains with. Type `bin/ct-ng` to launch the crosstool menu.

Building a toolchain for BeagleBone Black

Crosstool-NG can build many different combinations of toolchains. To make the initial configuration easier, it comes with a set of samples that cover many of the common use cases. Use `bin/ct-ng list-samples` to generate the list.

The BeagleBone Black has a TI AM335x SoC, which contains an ARM Cortex A8 core and a VFPv3 floating-point unit. Since the BeagleBone Black has plenty of RAM and storage, we can use `glibc` as the C library. The closest sample is `arm-cortex_a8-linux-gnueabi`.

You can see the default configuration by prefixing the name with `show-`, as demonstrated here:

```
$ bin/ct-ng show-arm-cortex_a8-linux-gnueabi
[G...]   arm-cortex_a8-linux-gnueabi
    Languages       : C,C++
    OS              : linux-4.20.8
    Binutils        : binutils-2.32
    Compiler        : gcc-8.3.0
    C library       : glibc-2.29
    Debug tools     : duma-2_5_15 gdb-8.2.1 ltrace-0.7.3
strace-4.26
    Companion libs  : expat-2.2.6 gettext-0.19.8.1 gmp-6.1.2
isl-0.20 libelf-0.8.13 libiconv-1.15 mpc-1.1.0 mpfr-4.0.2
ncurses-6.1 zlib-1.2.11
    Companion tools :
```

This is a close match with our requirements, except that it uses the `eabi` binary interface, which passes floating-point arguments in integer registers. We would prefer to use hardware floating point registers for that purpose because it would speed up function calls that have `float` and `double` parameter types. You can change the configuration later, so for now you should select this target configuration:

```
$ bin/ct-ng arm-cortex_a8-linux-gnueabi
```

At this point, you can review the configuration and make changes using the configuration menu command `menuconfig`:

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

The menu system is based on the Linux kernel `menuconfig`, and so navigation of the user interface will be familiar to anyone who has configured a kernel. If not, refer to *Chapter 4*, *Configuring and Building the Kernel*, for a description of `menuconfig`.

There are three configuration changes that I would recommend you make at this point:

- In **Paths and misc options**, disable **Render the toolchain read-only** (`CT_PREFIX_DIR_RO`).
- In **Target options** | **Floating point**, select **hardware** (**FPU**) (`CT_ARCH_FLOAT_HW`).
- In **Target options**, enter `neon` for **Use specific FPU**.

The first is necessary if you want to add libraries to the toolchain after it has been installed, which I describe later, in the *Linking with libraries* section. The second selects the `eabihf` binary interface for the reasons discussed earlier. The third is needed to build the Linux kernel successfully. The names in parentheses are the configuration labels stored in the configuration file. When you have made the changes, exit the `menuconfig` menu and save the configuration as one does.

Now you can use crosstool-NG to get, configure, and build the components according to your specification, by typing the following command:

```
$ bin/ct-ng build
```

The build will take about half an hour, after which you will find your toolchain is present in `~/x-tools/arm-cortex_a8-linux-gnueabihf`.

Next, let's build a toolchain that targets QEMU.

Building a toolchain for QEMU

On the QEMU target, you will be emulating an ARM-versatile PB evaluation board that has an ARM926EJ-S processor core, which implements the ARMv5TE instruction set. You need to generate a crosstool-NG toolchain that matches the specification. The procedure is very similar to the one for the BeagleBone Black.

You begin by running `bin/ct-ng list-samples` to find a good base configuration to work from. There isn't an exact fit, so use a generic target, `arm-unknown-linux-gnueabi`. You select it as shown, running `distclean` first to make sure that there are no artifacts left over from a previous build:

```
$ bin/ct-ng distclean
$ bin/ct-ng arm-unknown-linux-gnueabi
```

As with the BeagleBone Black, you can review the configuration and make changes using the configuration menu command `bin/ct-ng menuconfig`. There is only one change necessary:

- In **Paths and misc options**, disable **Render the toolchain read-only** (`CT_PREFIX_DIR_RO`).

Now, build the toolchain with the command shown here:

```
$ bin/ct-ng build
```

As before, the build will take about half an hour. The toolchain will be installed in `~/x-tools/arm-unknown-linux-gnueabi`.

You will need a working cross toolchain to complete the exercises in the next section.

Anatomy of a toolchain

To get an idea of what is in a typical toolchain, I want to examine the crosstool-NG toolchain you have just created. The examples use the ARM Cortex A8 toolchain created for the BeagleBone Black, which has the prefix `arm-cortex_a8-linux-gnueabihf-`. If you built the ARM926EJ-S toolchain for the QEMU target, then the prefix will be `arm-unknown-linux-gnueabi` instead.

The ARM Cortex A8 toolchain is in the directory `~/x-tools/arm-cortex_a8-linux-gnueabihf/bin`. In there, you will find the cross compiler, `arm-cortex_a8-linux-gnueabihf-gcc`. To make use of it, you need to add the directory to your path using the following command:

```
$ PATH=~/x-tools/arm-cortex_a8-linux-gnueabihf/bin:$PATH
```

Now you can take a simple `helloworld` program, which in the C language looks like this:

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    printf ("Hello, world!\n");
    return 0;
}
```

You compile it like this:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -o helloworld
```

You can confirm that it has been cross-compiled by using the `file` command to print the type of the file:

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, EABI5 version 1
(SYSV), dynamically linked, interpreter /lib/ld-linux-armhf.
so.3, for GNU/Linux 4.20.8, with debug_info, not stripped
```

Now that you've verified that your cross compiler works, let's take a closer look at it.

Finding out about your cross compiler

Imagine that you have just received a toolchain and that you would like to know more about how it was configured. You can find out a lot by querying `gcc`. For example, to find the version, you use `--version`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc --version
arm-cortex_a8-linux-gnueabihf-gcc (crosstool-NG 1.24.0) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.
```

To find how it was configured, use `-v`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=arm-cortex_a8-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/home/frank/x-tools/arm-cortex_a8-linux-
gnueabihf/libexec/gcc/arm-cortex_a8-linux-gnueabihf/8.3.0/
lto-wrapper
Target: arm-cortex_a8-linux-gnueabihf
Configured with: /home/frank/crosstool-ng/.build/arm-cortex_
a8-linux-gnueabihf/src/gcc/configure --build=x86_64-build_
pc-linux-gnu --host=x86_64-build_pc-linux-gnu --target=arm-
cortex_a8-linux-gnueabihf --prefix=/home/frank/x-tools/
arm-cortex_a8-linux-gnueabihf --with-sysroot=/home/frank/x-
tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-linux-
gnueabihf/sysroot --enable-languages=c,c++ --with-cpu=cortex-a8
--with-float=hard --with-pkgversion='crosstool-NG 1.24.0'
--enable-__cxa_atexit --disable-libmudflap --disable-libgomp
--disable-libssp --disable-libquadmath --disable-libquadmath-
support --disable-libsanitizer --disable-libmpx --with-gmp=/
home/frank/crosstool-ng/.build/arm-cortex_a8-linux-gnueabihf/
buildtools --with-mpfr=/home/frank/crosstool-ng/.build/
arm-cortex_a8-linux-gnueabihf/buildtools --with-mpc=/home/
frank/crosstool-ng/.build/arm-cortex_a8-linux-gnueabihf/
buildtools --with-isl=/home/frank/crosstool-ng/.build/
arm-cortex_a8-linux-gnueabihf/buildtools --enable-lto --with-
host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic
-lm' --enable-threads=posix --enable-target-optspace --enable-
plugin --enable-gold --disable-nls --disable-multilib --with-
local-prefix=/home/frank/x-tools/arm-cortex_a8-linux-gnueabihf/
arm-cortex_a8-linux-gnueabihf/sysroot --enable-long-long
Thread model: posix
gcc version 8.3.0 (crosstool-NG 1.24.0)
```

There is a lot of output there, but the interesting things to note are the following:

- `--with-sysroot=/home/frank/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-linux-gnueabihf/sysroot`: This is the default `sysroot` directory; see the following section for an explanation.
- `--enable-languages=c,c++`: Using this, we have both C and C++ languages enabled.

- `--with-cpu=cortex-a8`: The code is generated for an ARM Cortex A8 core.
- `--with-float=hard`: Generates opcodes for the floating-point unit and uses the VFP registers for parameters.
- `--enable-threads=posix`: This enables the POSIX threads.

These are the default settings for the compiler. You can override most of them on the `gcc` command line. For example, if you want to compile for a different CPU, you can override the configured setting, `--with-cpu`, by adding `-mcpu` to the command line, as follows:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -mcpu=cortex-a5 \
helloworld.c \
-o helloworld
```

You can print out the range of architecture-specific options available using `--target-help`, as follows:

```
$ arm-cortex_a8-linux-gnueabihf-gcc --target-help
```

You may be wondering whether it matters that you get the configuration exactly right at this point, since you can always change it as shown here. The answer depends on the way you anticipate using it. If you plan to create a new toolchain for each target, then it makes sense to set everything up at the beginning, because it will reduce the risk of getting it wrong later on. Jumping ahead a little to *Chapter 6*, *Selecting a Build System*, I call this the Buildroot philosophy. If, on the other hand, you want to build a toolchain that is generic and you are prepared to provide the correct settings when you build for a particular target, then you should make the base toolchain generic, which is the way the Yocto Project handles things. The preceding examples follow the Buildroot philosophy.

The sysroot, library, and header files

The toolchain `sysroot` is a directory that contains subdirectories for libraries, header files, and other configuration files. It can be set when the toolchain is configured through `--with-sysroot=`, or it can be set on the command line using `--sysroot=`. You can see the location of the default `sysroot` by using `-print-sysroot`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot
/home/frank/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_
a8-linux-gnueabihf/sysroot
```

You will find the following subdirectories in `sysroot`:

- `lib`: Contains the shared objects for the C library and the dynamic linker/loader, `ld-linux`
- `usr/lib`: The static library archive files for the C library, and any other libraries that may be installed subsequently
- `usr/include`: Contains the headers for all the libraries
- `usr/bin`: Contains the utility programs that run on the target, such as the `ldd` command
- `usr/share`: Used for localization and internationalization
- `sbin`: Provides the `ldconfig` utility, used to optimize library loading paths

Plainly, some of these are needed on the development host to compile programs, and others, for example, the shared libraries and `ld-linux`, are needed on the target at runtime.

Other tools in the toolchain

Below is a list of commands to invoke the various other components of a GNU toolchain, together with a brief description:

- `addr2line`: Converts program addresses into filenames and numbers by reading the debug symbol tables in an executable file. It is very useful when decoding addresses printed out in a system crash report.
- `ar`: The archive utility is used to create static libraries.
- `as`: This is the GNU assembler.
- `c++filt`: This is used to demangle C++ and Java symbols.
- `cpp`: This is the C preprocessor and is used to expand `#define`, `#include`, and other similar directives. You seldom need to use this by itself.
- `elfedit`: This is used to update the ELF header of the ELF files.
- `g++`: This is the GNU C++ frontend, which assumes that source files contain C++ code.
- `gcc`: This is the GNU C frontend, which assumes that source files contain C code.
- `gcov`: This is a code coverage tool.
- `gdb`: This is the GNU debugger.

- `gprof`: This is a program profiling tool.
- `ld`: This is the GNU linker.
- `nm`: This lists symbols from object files.
- `objcopy`: This is used to copy and translate object files.
- `objdump`: This is used to display information from object files.
- `ranlib`: This creates or modifies an index in a static library, making the linking stage faster.
- `readelf`: This displays information about files in ELF object format.
- `size`: This lists section sizes and the total size.
- `strings`: This displays strings of printable characters in files.
- `strip`: This is used to strip an object file of debug symbol tables, thus making it smaller. Typically, you would strip all the executable code that is put onto the target.

We will now switch gears from command-line tools and return to the topic of the C library.

Looking at the components of the C library

The C library is not a single library file. It is composed of four main parts that together implement the POSIX API:

- `libc`: The main C library that contains the well-known POSIX functions such as `printf`, `open`, `close`, `read`, `write`, and so on
- `libm`: Contains math functions such as `cos`, `exp`, and `log`
- `libpthread`: Contains all the POSIX thread functions with names beginning with `pthread_`
- `librt`: Has the real-time extensions to POSIX, including shared memory and asynchronous I/O

The first one, `libc`, is always linked in but the others have to be explicitly linked with the `-l` option. The parameter to `-l` is the library name with `lib` stripped off. For example, a program that calculates a sine function by calling `sin()` would be linked with `libm` using `-lm`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc myprog.c -o myprog -lm
```

You can verify which libraries have been linked in this or any other program by using the `readelf` command:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep
"Shared library"
 0x00000001 (NEEDED)                Shared library: [libm.so.6]
 0x00000001 (NEEDED)                Shared library: [libc.so.6]
```

Shared libraries need a runtime linker, which you can expose using this:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep
"program interpreter"
    [Requesting program interpreter: /lib/ld-linux-armhf.so.3]
```

This is so useful that I have a script file named `list-libs`, which you will find in the book code archive in `MELP/list-libs`. It contains the following commands:

```
#!/bin/sh
${CROSS_COMPILE}readelf -a $1 | grep "program interpreter"
${CROSS_COMPILE}readelf -a $1 | grep "Shared library"
```

There are other library files we can link to other than the four components of the C library. We will look at how to do that in the next section.

Linking with libraries – static and dynamic linking

Any application you write for Linux, whether it be in C or C++, will be linked with the `libc` C library. This is so fundamental that you don't even have to tell `gcc` or `g++` to do it because it always links `libc`. Other libraries that you may want to link with have to be explicitly named through the `-l` option.

The library code can be linked in two different ways: statically, meaning that all the library functions your application calls and their dependencies are pulled from the library archive and bound into your executable; and dynamically, meaning that references to the library files and functions in those files are generated in the code but the actual linking is done dynamically at runtime. You will find the code for the examples that follow in the book code archive in `MELP/Chapter02/library`.

We'll start with static linking.

Static libraries

Static linking is useful in a few circumstances. For example, if you are building a small system that consists of only BusyBox and some script files, it is simpler to link BusyBox statically and avoid having to copy the runtime library files and linker. It will also be smaller because you only link in the code that your application uses rather than supplying the entire C library. Static linking is also useful if you need to run a program before the filesystem that holds the runtime libraries is available.

You can link all the libraries statically by adding `-static` to the command line:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -static helloworld.c -o
helloworld-static
```

You will note that the size of the binary increases dramatically:

```
$ ls -l
total 4060
-rwxrwxr-x 1 frank frank   11816 Oct 23 15:45 helloworld
-rw-rw-r-- 1 frank frank     123 Oct 23 15:35 helloworld.c
-rwxrwxr-x 1 frank frank 4140860 Oct 23 16:00 helloworld-static
```

Static linking pulls code from a library archive, usually named `lib[name].a`. In the preceding case, it is `libc.a`, which is in `[sysroot]/usr/lib`:

```
$ export SYSROOT=$(arm-cortex_a8-linux-gnueabihf-gcc -print-
sysroot)
$ cd $SYSROOT
$ ls -l usr/lib/libc.a
-rw-r--r-- 1 frank frank 31871066 Oct 23 15:16 usr/lib/libc.a
```

Note that the syntax export `SYSROOT=$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)` places the path to the `sysroot` in the shell variable, `SYSROOT`, which makes the example a little clearer.

Creating a static library is as simple as creating an archive of object files using the `ar` command. If I have two source files named `test1.c` and `test2.c`, and I want to create a static library named `libtest.a`, then I would do the following:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -c test1.c
$ arm-cortex_a8-linux-gnueabihf-gcc -c test2.c
$ arm-cortex_a8-linux-gnueabihf-ar rc libtest.a test1.o test2.o
$ ls -l
```

```
total 24
-rw-rw-r-- 1 frank frank 2392 Oct 9 09:28 libtest.a
-rw-rw-r-- 1 frank frank 116 Oct 9 09:26 test1.c
-rw-rw-r-- 1 frank frank 1080 Oct 9 09:27 test1.o
-rw-rw-r-- 1 frank frank 121 Oct 9 09:26 test2.c
-rw-rw-r-- 1 frank frank 1088 Oct 9 09:27 test2.o
```

Then I could link `libtest` into my `helloworld` program, using this:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -ltest \
-L../libs -I../libs -o helloworld
```

Now let's rebuild the same program using dynamic linking.

Shared libraries

A more common way to deploy libraries is as shared objects that are linked at runtime, which makes more efficient use of storage and system memory, since only one copy of the code needs to be loaded. It also makes it easy to update the library files without having to relink all the programs that use them.

The object code for a shared library must be position-independent, so that the runtime linker is free to locate it in memory at the next free address. To do this, add the `-fPIC` parameter to `gcc`, and then link it using the `-shared` option:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -fPIC -c test1.c
$ arm-cortex_a8-linux-gnueabihf-gcc -fPIC -c test2.c
$ arm-cortex_a8-linux-gnueabihf-gcc -shared -o libtest.so
test1.o test2.o
```

This creates the shared library, `libtest.so`. To link an application with this library, you add `-ltest`, exactly as in the static case mentioned in the preceding section, but this time the code is not included in the executable. Instead, there is a reference to the library that the runtime linker will have to resolve:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -ltest \
-L../libs -I../libs -o helloworld
$ MELP/list-libs helloworld
    [Requesting program interpreter: /lib/ld-linux-armhf.so.3]
 0x00000001 (NEEDED)             Shared library: [libtest.so.6]
 0x00000001 (NEEDED)             Shared library: [libc.so.6]
```

The runtime linker for this program is `/lib/ld-linux-armhf.so.3`, which must be present in the target's filesystem. The linker will look for `libtest.so` in the default search path: `/lib` and `/usr/lib`. If you want it to look for libraries in other directories as well, you can place a colon-separated list of paths in the `LD_LIBRARY_PATH` shell variable:

```
$ export LD_LIBRARY_PATH=/opt/lib:/opt/usr/lib
```

Because shared libraries are separate from the executables they link to, we need to be aware of their versions when deploying them.

Understanding shared library version numbers

One of the benefits of shared libraries is that they can be updated independently of the programs that use them.

Library updates are of two types:

- Those that fix bugs or add new functions in a backward-compatible way
- Those that break compatibility with existing applications

GNU/Linux has a versioning scheme to handle both these cases.

Each library has a release version and an interface number. The release version is simply a string that is appended to the library name; for example, the JPEG image library `libjpeg` is currently at release `8.2.2` and so the library is named `libjpeg.so.8.2.2`. There is a symbolic link named `libjpeg.so` to `libjpeg.so.8.2.2`, so that when you compile a program with `-ljpeg`, you link with the current version. If you install version `8.2.3`, the link is updated, and you will link with that one instead.

Now suppose that version `9.0.0` comes along and that breaks the backward compatibility. The link from `libjpeg.so` now points to `libjpeg.so.9.0.0`, so that any new programs are linked with the new version, possibly throwing compile errors when the interface to `libjpeg` changes, which the developer can fix.

Any programs on the target that are not recompiled are going to fail in some way, because they are still using the old interface. This is where an object known as the **soname** helps. The soname encodes the interface number when the library was built and is used by the runtime linker when it loads the library. It is formatted as `<library name>.so.<interface number>`. For `libjpeg.so.8.2.2`, the soname is `libjpeg.so.8` because the interface number when that `libjpeg` shared library was built is `8`:

```
$ readelf -a /usr/lib/x86_64-linux-gnu/libjpeg.so.8.2.2 \
| grep SONAME
 0x000000000000000e (SONAME)    Library soname: [libjpeg.so.8]
```

Any program compiled with it will request `libjpeg.so.8` at runtime, which will be a symbolic link on the target to `libjpeg.so.8.2.2`. When version `9.0.0` of `libjpeg` is installed, it will have a soname of `libjpeg.so.9`, and so it is possible to have two incompatible versions of the same library installed on the same system. Programs that were linked with `libjpeg.so.8.*.*` will load `libjpeg.so.8`, and those linked with `libjpeg.so.9.*.*` will load `libjpeg.so.9`.

This is why, when you look at the directory listing of `/usr/lib/x86_64-linux-gnu/libjpeg*`, you find these four files:

- `libjpeg.a`: This is the library archive used for static linking.
- `libjpeg.so -> libjpeg.so.8.2.2`: This is a symbolic link, used for dynamic linking.
- `libjpeg.so.8 -> libjpeg.so.8.2.2`: This is a symbolic link, used when loading the library at runtime.
- `libjpeg.so.8.2.2`: This is the actual shared library, used at both compile time and runtime.

The first two are only needed on the host computer for building and the last two are needed on the target at runtime.

While you can invoke the various GNU cross-compilation tools directly from the command line, this technique does not scale beyond toy examples such as `helloworld`. To really be effective at cross-compiling, we need to combine a cross toolchain with a build system.

The art of cross-compiling

Having a working cross toolchain is the starting point of a journey, not the end of it. At some point, you will want to begin cross-compiling the various tools, applications, and libraries that you need on your target. Many of them will be open source packages, each of which has its own method of compiling and its own peculiarities.

There are some common build systems, including the following:

- Pure makefiles, where the toolchain is usually controlled by the `make` variable `CROSS_COMPILE`
- The GNU build system known as **Autotools**
- **CMake** (`https://cmake.org`)

Both Autotools and makefiles are needed to build even a basic embedded Linux system. CMake is cross-platform and has seen increased adoption over the years especially among the C++ community. In this section, we will cover all three build tools.

Simple makefiles

Some important packages are very simple to cross-compile, including the Linux kernel, the U-Boot bootloader, and BusyBox. For each of these, you only need to put the toolchain prefix in the `make` variable `CROSS_COMPILE`, for example, `arm-cortex_a8-linux-gnueabi-`. Note the trailing dash `-`.

So, to compile BusyBox, you would type this:

```
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

Or, you can set it as a shell variable:

```
$ export CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
$ make
```

In the case of U-Boot and Linux, you also have to set the `make` variable `ARCH` to one of the machine architectures they support, which I will cover in *Chapter 3*, *All About Bootloaders*, and *Chapter 4*, *Configuring and Building the Kernel*.

Both Autotools and CMake can generate makefiles. Autotools only generates makefiles whereas CMake supports other ways of building projects depending on which platform(s) we are targeting (strictly Linux in our case). Let's look at cross-compiling with Autotools first.

Autotools

The name Autotools refers to a group of tools that are used as the build system in many open source projects. The components, together with the appropriate project pages, are as follows:

- GNU Autoconf (`https://www.gnu.org/software/autoconf/autoconf.html`)
- GNU Automake (`https://www.gnu.org/savannah-checkouts/gnu/automake/`)
- GNU Libtool (`https://www.gnu.org/software/libtool/libtool.html`)
- Gnulib (`https://www.gnu.org/software/gnulib/`)

The role of Autotools is to smooth over the differences between the different types of systems that the package may be compiled for, accounting for different versions of compilers, different versions of libraries, different locations of header files, and dependencies with other packages.

Packages that use Autotools come with a script named `configure` that checks dependencies and generates makefiles according to what it finds. The `configure` script may also give you the opportunity to enable or disable certain features. You can find the options on offer by running `./configure --help`.

To configure, build and install a package for the native operating system, you would typically run the following three commands:

```
$ ./configure
$ make
$ sudo make install
```

Autotools is able to handle cross-development as well. You can influence the behavior of the configured script by setting these shell variables:

- `CC`: The C compiler command.
- `CFLAGS`: Additional C compiler flags.
- `CXX`: The C++ compiler command.
- `CXXFLAGS`: Additional C++ compiler flags.
- `LDFLAGS`: Additional linker flags; for example, if you have libraries in a non-standard directory `<lib dir>`, you would add it to the library search path by adding `-L<lib dir>`.
- `LIBS`: Contains a list of additional libraries to pass to the linker; for instance, `-lm` for the math library.
- `CPPFLAGS`: Contains C/C++ preprocessor flags; for example, you would add `-I<include dir>` to search for headers in a non-standard directory `<include dir>`.
- `CPP`: The C preprocessor to use.

Sometimes it is sufficient to set only the `CC` variable, as follows:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc ./configure
```

At other times, that will result in an error like this:

```
[…]
checking for suffix of executables...
checking whether we are cross compiling... configure: error: in
'/home/frank/sqlite-autoconf-3330000':
configure: error: cannot run C compiled programs.
If you meant to cross compile, use '--host'.
See 'config.log' for more details
```

The reason for the failure is that `configure` often tries to discover the capabilities of the toolchain by compiling snippets of code and running them to see what happens, which cannot work if the program has been cross-compiled.

> **Important note**
>
> Pass `--host=<host>` to `configure` when you are cross-compiling so that `configure` searches your system for the cross-compiling toolchain targeting the specified `<host>` platform. That way, `configure` does not try to run snippets of non-native code as part of the configuration step.

Autotools understands three different types of machines that may be involved when compiling a package:

- **Build**: The computer that builds the package, which defaults to the current machine.
- **Host**: The computer the program will run on. For a native compile, this is left blank and it defaults to be the same computer as Build. When you are cross-compiling, set it to be the tuple of your toolchain.
- **Target**: The computer the program will generate code for. You would set this when building a cross compiler.

So, to cross-compile, you just need to override the host, as follows:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf
```

One final thing to note is that the default install directory is `<sysroot>/usr/local/*`. You would usually install it in `<sysroot>/usr/*` so that the header files and libraries would be picked up from their default locations.

The complete command to configure a typical Autotools package is as follows:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

Let's dive deeper into Autotools and use it to cross-compile a popular library.

An example – SQLite

The SQLite library implements a simple relational database and is quite popular on embedded devices. You begin by getting a copy of SQLite:

```
$ wget http://www.sqlite.org/2020/sqlite-autoconf-3330000.tar.
gz
$ tar xf sqlite-autoconf-3330000.tar.gz
$ cd sqlite-autoconf-3330000
```

Next, run the `configure` script:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

That seems to work! If it had failed, there would be error messages printed to the Terminal and recorded in `config.log`. Note that several makefiles have been created, so now you can build it:

```
$ make
```

Finally, you install it into the toolchain directory by setting the `make` variable `DESTDIR`. If you don't, it will try to install it into the host computer's `/usr` directory, which is not what you want:

```
$ make DESTDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-
sysroot) install
```

You may find that the final command fails with a file permissions error. A crosstool-NG toolchain is read-only by default, which is why it is useful to set `CT_PREFIX_DIR_RO` to `y` when building it. Another common problem is that the toolchain is installed in a system directory, such as `/opt` or `/usr/local`. In that case, you will need `root` permissions when running the install.

After installing, you should find that various files have been added to your toolchain:

- `<sysroot>/usr/bin: sqlite3`: This is a command-line interface for SQLite that you can install and run on the target.
- `<sysroot>/usr/lib`: `libsqlite3.so.0.8.6`, `libsqlite3.so.0`, `libsqlite3.so`, `libsqlite3.la`, `libsqlite3.a`: These are the shared and static libraries.
- `<sysroot>/usr/lib/pkgconfig: sqlite3.pc`: This is the package configuration file, as described in the following section.
- `<sysroot>/usr/lib/include: sqlite3.h`, `sqlite3ext.h`: These are the header files.
- `<sysroot>/usr/share/man/man1: sqlite3.1`: This is the manual page.

Now you can compile programs that use `sqlite3` by adding `-lsqlite3` at the link stage:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -lsqlite3 sqlite-test.c -o
sqlite-test
```

Here, `sqlite-test.c` is a hypothetical program that calls SQLite functions. Since `sqlite3` has been installed into the `sysroot`, the compiler will find the header and library files without any problem. If they had been installed elsewhere, you would have had to add `-L<lib dir>` and `-I<include dir>`.

Naturally, there will be runtime dependencies as well, and you will have to install the appropriate files into the target directory as described in *Chapter 5*, *Building a Root Filesystem*.

In order to cross-compile a library or package, its dependencies first need to be cross-compiled. Autotools relies on a utility called `pkg-config` to gather vital information about packages cross-compiled by Autotools.

Package configuration

Tracking package dependencies is quite complex. The package configuration utility `pkg-config` (`https://www.freedesktop.org/wiki/Software/pkg-config/`) helps track which packages are installed and which compile flags each package needs by keeping a database of Autotools packages in `[sysroot]/usr/lib/pkgconfig`. For instance, the one for SQLite3 is named `sqlite3.pc` and contains essential information needed by other packages that need to make use of it:

```
$ cat $(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)/usr/
lib/pkgconfig/sqlite3.pc
# Package Information for pkg-config

prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: SQLite
Description: SQL database engine
Version: 3.33.0
Libs: -L${libdir} -lsqlite3
Libs.private: -lm -ldl -lpthread
Cflags: -I${includedir}
```

You can use `pkg-config` to extract information in a form that you can feed straight to `gcc`. In the case of a library like `libsqlite3`, you want to know the library name (`--libs`) and any special C flags (`--cflags`):

```
$ pkg-config sqlite3 --libs --cflags
Package sqlite3 was not found in the pkg-config search path.
Perhaps you should add the directory containing 'sqlite3.pc'
to the PKG_CONFIG_PATH environment variable
No package 'sqlite3' found
```

Oops! That failed because it was looking in the host's `sysroot` and the development package for `libsqlite3` has not been installed on the host. You need to point it at the `sysroot` of the target toolchain by setting the `PKG_CONFIG_LIBDIR` shell variable:

```
$ export PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabihf-gcc \
-print-sysroot)/usr/lib/pkgconfig
$ pkg-config sqlite3 --libs --cflags
-lsqlite3
```

Now the output is `-lsqlite3`. In this case, you knew that already, but generally you wouldn't, so this is a valuable technique. The final commands to compile would be the following:

```
$ export PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabihf-gcc \
-print-sysroot)/usr/lib/pkgconfig
$ arm-cortex_a8-linux-gnueabihf-gcc $(pkg-config sqlite3
--cflags --libs) \
sqlite-test.c -o sqlite-test
```

Many configure scripts read the information generated by `pkg-config`. This can lead to errors when cross-compiling as we shall see next.

Problems with cross-compiling

`sqlite3` is a well-behaved package and cross-compiles nicely, but not all packages are the same. Typical pain points include the following:

- Home-grown build systems for libraries such as `zlib` that have a `configure` script that does not behave like the Autotools `configure` described in the previous section
- Configure scripts that read `pkg-config` information, headers, and other files from the host, disregarding the `--host` override
- Scripts that insist on trying to run cross-compiled code

Each case requires careful analysis of the error and additional parameters to the `configure` script to provide the correct information, or patches to the code to avoid the problem altogether. Bear in mind that one package may have many dependencies, especially with programs that have a graphical interface using GTK or Qt, or that handle multimedia content. As an example, `mplayer`, which is a popular tool for playing multimedia content, has dependencies on over 100 libraries. It would take weeks of effort to build them all.

Therefore, I would not recommend manually cross-compiling components for the target in this way, except when there is no alternative or the number of packages to build is small. A much better approach is to use a build tool such as Buildroot or the Yocto Project or avoid the problem altogether by setting up a native build environment for your target architecture. Now you can see why distributions such as Debian are always compiled natively.

CMake

CMake is more of a meta build system in the sense that it relies on an underlying platform's native tools to build software. On Windows, CMake can generate project files for Microsoft Visual Studio and on macOS, it can generate project files for Xcode. Integrating with the principal IDEs for each of the major platforms is no simple task and explains the success of CMake as the leading cross-platform build system solution. CMake also runs on Linux, where it can be used in conjunction with a cross-compiling toolchain of your choice.

To configure, build, and install a package for a native Linux operating system, run the following commands:

```
$ cmake .
$ make
$ sudo make install
```

On Linux, the native build tool is GNU `make` so CMake generates makefiles by default for us to build with. Oftentimes, we want to perform out-of-source builds so that object files and other build artifacts remain separate from source files.

To configure an out-of-source build in a subdirectory named `_build`, run the following commands:

```
$ mkdir _build
$ cd _build
$ cmake ..
```

This will generate the makefiles inside a `_build` subdirectory within the project directory where the `CMakeLists.txt` is located. The `CMakeLists.txt` file is the CMake equivalent of the `configure` script for Autotools-based projects.

We can then build the project out-of-source from inside the `_build` directory and install the package just as before:

```
$ make
$ sudo make install
```

CMake uses absolute paths so the `_build` subdirectory cannot be copied or moved once the makefiles have been generated or any subsequent `make` step will likely fail. Note that CMake defaults to installing packages into system directories such as `/usr/bin` even for out-of-source builds.

To generate the makefiles so that `make` installs the package in the `_build` subdirectory, replace the previous `cmake` command with the following:

```
$ cmake .. -D CMAKE_INSTALL_PREFIX=../_build
```

We no longer need to preface `make install` with `sudo` because we do not need elevated permissions to copy the package files into the `_build` directory.

Similarly, we can use another CMake command-line option to generate makefiles for cross-compilation:

```
$ cmake .. -D CMAKE_C_COMPILER="/usr/local/share/x-tools/
arm-cortex_a8-linux-gnueabihf-gcc"
```

But the best practice for cross-compiling with CMake is to create a toolchain file that sets `CMAKE_C_COMPILER` and `CMAKE_CXX_COMPILER` in addition to other relevant variables for targeting embedded Linux.

CMake works best when we design our software in a modular way by enforcing well-defined API boundaries between libraries and components.

Here are some key terms that come up time and time again in CMake:

- `target`: A software component such as a library or executable.
- `properties`: Include the source files, compiler options, and linked libraries needed to build a target.
- `package`: A CMake file that configures an external target for building just as if it was defined within your `CMakeLists.txt` itself.

For example, if we had a CMake-based executable named `dummy` that needed to take a dependency on SQLite, we could define the following `CMakeLists.txt`:

```
cmake_minimum_required (VERSION 3.0)
project (Dummy)
add_executable(dummy dummy.c)
find_package (SQLite3)
target_include_directories(dummy PRIVATE ${SQLITE3_INCLUDE_
DIRS})
target_link_libraries (dummy PRIVATE ${SQLITE3_LIBRARIES})
```

The `find_package` command searches for a package (`SQLite3` in this case) and imports it so that the external target can be added as a dependency to the `dummy` executable's list of `target_link_libraries` for linking.

CMake comes with numerous finders for popular C and C++ packages including OpenSSL, Boost, and protobuf, making native development much more productive than if we were to use just pure makefiles.

The `PRIVATE` qualifier prevents details such as headers and flags from leaking outside of the `dummy` target. Using `PRIVATE` makes more sense when the target being built is a library instead of an executable. Think of targets as modules and attempt to minimize their exposed surface areas when using CMake to define your own targets. Only employ the `PUBLIC` qualifier when absolutely necessary and utilize the `INTERFACE` qualifier for header-only libraries.

Model your application as a dependency graph with edges between targets. This graph should not only include the libraries that your application links to directly but any transitive dependencies as well. For best results, remove any cycles or other unnecessary independencies seen in the graph. It is often best to perform this exercise before you start coding. A little planning can make the difference between a clean, easily maintainable `CMakeLists.txt` and an inscrutable mess that nobody wants to touch.

Summary

The toolchain is always your starting point; everything that follows from that is dependent on having a working, reliable toolchain.

You may start with nothing but a toolchain—perhaps built using crosstool-NG or downloaded from Linaro—and use it to compile all the packages that you need on your target. Or you may obtain the toolchain as part of a distribution generated from source code using a build system such as Buildroot or the Yocto Project. Beware of toolchains or distributions that are offered to you for free as part of a hardware package; they are often poorly configured and not maintained.

Once you have a toolchain, you can use it to build the other components of your embedded Linux system. In the next chapter, you will learn about the bootloader, which brings your device to life and begins the boot process. We will use the toolchain we built in this chapter to build a working bootloader for the BeagleBone Black.