2

A BRIEF INTRODUCTION TO THE GNU AUTOTOOLS

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
—T.S. Eliot, "Quartet No. 4: Little Gidding"

As stated in the preface to this book, the purpose of the GNU Autotools is to make life simpler for the end user, not the maintainer. Nevertheless, using the Autotools will make your job as a project maintainer easier in the long run, although maybe not for the reasons you suspect. The Autotools framework is as simple as it can be, given the functionality it provides. The real purpose of the Autotools is twofold: it serves the needs of your users, and it makes your project incredibly portable—even to systems on which you've never tested, installed, or built your code.

Throughout this book, I will often use the term *Autotools*, although you won't find a package in the GNU archives with this label. I use this term to signify the following three GNU projects, which are considered by the community to be part of the GNU build system:

- Autoconf, which is used to generate a configuration script for a project
- Automake, which is used to simplify the process of creating consistent and functional makefiles
- Libtool, which provides an abstraction for the portable creation of shared libraries

Other build tools, such as the open source projects CMake and SCons, attempt to provide the same functionality as the Autotools but in a more user-friendly manner. However, because these tools attempt to hide much of their complexity behind GUI interfaces and script builders, they actually end up being less functional, and more difficult to manage, because the build system is not as transparent. In the final analysis, this transparency is what makes the Autotools both simpler to use and simpler to understand. Initial frustration with the Autotools, therefore, comes not from their complexity—for they are truly very simple—but from their extensive use of less well understood tools and subsystems, such as the Linux command shell (Bash), the `make` utility, and the M4 macro processor and accompanying macro libraries. Indeed, the meta-language provided by Automake is so simple it can be entirely digested and comprehended within a few hours of perusing the manual (though the ramifications of this meta-language may take a bit longer to thoroughly internalize).

Who Should Use the Autotools?

If you're writing open source software that targets Unix or Linux systems, you should absolutely be using the GNU Autotools, and even if you're writing proprietary software for Unix or Linux systems, you'll still benefit significantly from using them. The Autotools provide you with a build environment that allows your project to build successfully on future versions or distributions with virtually no changes to the build scripts. This is useful even if you only intend to target a single Linux distribution, because—let's be honest—you really *can't* know in advance whether or not your company will want your software to run on other platforms in the future.

When Should You Not Use the Autotools?

About the only time it makes sense not to use the Autotools is when you're writing software that will *only* run on non-Unix platforms, such as Microsoft Windows.

Autotools support for Windows requires an Msys[1] environment in order to work correctly, because Autoconf-generated configuration scripts are Bourne-shell scripts, and Windows doesn't provide a native Bourne shell.[2] Unix and Microsoft tools are just different enough in command line options and runtime characteristics that it's often simpler to use Windows ports of GNU tools, such as Cygwin, Msys2, or MinGW, to build Windows programs with an Autotools build system.

For these reasons, I'll focus mostly on using the Autotools on POSIX-compliant platforms. Nevertheless, if you're interested in trying out the Autotools on Windows, check out Chapter 17 for an in-depth overview.

NOTE *I'm not a typical Unix bigot. While I love Unix (and especially Linux), I also appreciate Windows for the areas in which it excels.[3] For Windows development, I highly recommend using Microsoft tools. The original reasons for using GNU tools to develop Windows programs are more or less academic nowadays because Microsoft has made the better part of its tools available for download at no cost. For download information, see Visual Studio Community at* https://visualstudio.microsoft.com/vs/express/.

Apple Platforms and Mac OS X

The Macintosh operating system has been POSIX compliant since 2007 when the "Leopard" release of macOS version 10 (OS X) was published. OS X is derived from NeXTSTEP/OpenStep, which is based on the Mach kernel, with parts taken from FreeBSD and NetBSD. As a POSIX-compliant operating system, OS X provides all the infrastructure required by the Autotools. The problems you'll encounter with OS X will most likely involve Apple's graphical user interface and package management systems, both of which are specific to the Mac.

The user interface presents the same issues you encounter when dealing with the X Window system on other Unix platforms, and then some. The primary difference is that the X Window system is used exclusively on most Unix systems, but macOS has its own graphical user interface called *Cocoa*. While the X Window system can be used on the Mac (Apple provides

1. See MinGW, Minimalist GNU for Windows at *http://www.mingw.org/* for more information on the Msys concept.

2. Windows 10 actually supports a Linux environment called the Windows Subsystem for Linux (WSL). The integration between the Windows host and the Linux subsystem is much tighter than that of, say, a virtual machine running Linux on a Windows host. It's well worth exploring if you're interested in running Linux but don't want to entirely give up your Windows applications. Be aware, however, that open source software programs built using the Autotools will not run as native Windows applications but will instead interface with the WSL kernel components. Perhaps these days the distinction simply isn't that important.

3. Hard-core gamers will agree with me, I'm sure. I wrote the original edition of this book on a laptop running Windows 7, but I used OpenOffice as my content editor, and I wrote the book's sample code on a 3GHz 64-bit dual-processor openSUSE 11.2 Linux workstation. Lately I've been running the Ubuntu-based Linux Mint distribution and using LibreOffice 5.3.

a window manager that makes X applications look a lot like native Cocoa apps), Mac programmers will sometimes wish to take full advantage of the native user interface features provided by the operating system.

The Autotools skirt the issue of package management differences between Unix platforms by simply ignoring them. Instead, they create packages that are little more than compressed source archives using the `tar` and `gzip` utilities, and they install and uninstall products from the `make` command line. The macOS package management system is an integral part of installing an application on an Apple system, and projects like Fink (*http://www.finkproject.org/*) and MacPorts (*http://www.macports.org/*) help make existing open source packages available on the Mac by providing simplified mechanisms for converting Autotools packages into installable Mac packages.

The bottom line is that the Autotools can be used quite effectively on Apple Macintosh systems running OS X or later, as long as you keep these caveats in mind.

The Choice of Language

Your choice of programming language is another important factor to consider when deciding whether to use the Autotools. Remember that the Autotools were designed by GNU people to manage GNU projects. In the GNU community, two factors determine the importance of a computer programming language:

- Are there any GNU packages written in the language?
- Does the GNU compiler tool set support the language?

Autoconf provides native support for the following languages based on these two criteria (by *native support*, I mean that Autoconf will compile, link, and run source-level feature checks in these languages):

- C
- C++
- Objective C
- Objective C++
- Fortran
- Fortran 77
- Erlang
- Go

Therefore, if you want to build a Java package, you can configure Automake to do so (as you'll see in Chapters 14 and 15), but you can't ask

Autoconf to compile, link, or run Java-based checks,[4] because Autoconf simply doesn't natively support Java. However, you can find Autoconf macros (which I will cover in more detail in later chapters) that enhance Autoconf's ability to manage the configuration process for projects written in Java.

The general feeling is that Java has plenty of its own build environments and tools that work very well (maven, for instance); therefore, adding full support for Java seems like a wasted effort. This is especially true since Java and its build tools are themselves highly portable—even to non-Unix/Linux platforms such as Windows.

Rudimentary support does exist in Automake for Java compilers and JVMs. I've used these features myself on projects, and they work well, as long as you don't try to push them too far.

If you're into Smalltalk, ADA, Modula, Lisp, Forth, or some other non-mainstream language, you're probably not too interested in porting your code to dozens of platforms and CPUs. However, if you *are* using a non-mainstream language and you're concerned about the portability of your build systems, consider adding support for your language to the Autotools yourself. This is not as daunting a task as you may think, and I guarantee that you'll be an Autotools expert when you're finished.[5]

Generating Your Package Build System

The GNU Autotools framework includes three main packages: Autoconf, Automake, and Libtool. The tools in these packages can depend on utilities and functionality from the gettext, M4, sed, make, and Perl packages, among others; however, the build systems generated by these packages rely only on a Bourne shell and the `make` utility.

With respect to the Autotools, it's important to distinguish between a *maintainer's* system and an *end user's* system. The design goals of the Autotools specify that an Autotools-generated build system should rely only on tools that are readily available and preinstalled on the end user's machine (assuming the end user's system has rudimentary support for building programs from source code). For example, the machine a maintainer uses to create distributions requires a Perl interpreter, but a machine on which an end user builds products from release distribution source archives should not require Perl (unless, of course, the project sources are written in Perl).

A corollary is that an end user's machine doesn't need to have the Autotools installed—an end user's system only requires a reasonably

4. This statement is not strictly true: I've seen third-party macros that use the Java virtual machine (JVM) to execute Java code within checks, but these are usually very special cases. None of the built-in Autoconf checks rely on a JVM in any way. Chapters 14 and 15 outline how you might use a JVM in an Autoconf check. Additionally, the portable nature of Java and the Java virtual machine specification make it fairly unlikely that you'll need to perform a Java-based Autoconf check in the first place.

5. For example, native Erlang support made it into the Autotools because members of the Erlang community thought it was important enough to add it themselves.

POSIX-compliant version of `make` and some variant of the Bourne shell that can execute the generated configuration script. And, of course, any package will also require compilers, linkers, and other tools needed to convert source files into executable binary programs, help files, and other runtime resources.

Configuration

Most developers understand the purpose of the `make` utility, but what's the point of `configure`? While Unix systems have followed the de facto standard Unix kernel interface for decades, most software has to stretch beyond these boundaries.

Originally, configuration scripts were hand-coded shell scripts designed to set environment variables based on platform-specific characteristics. They also allowed users to configure package options before running `make`. This approach worked well for decades, but as the number of Linux distributions and Unix-like systems grew, the variety of features and installation and configuration options exploded, so it became very difficult to write a decent portable configuration script. In fact, it was much more difficult to write a portable configuration script than it was to write makefiles for a new project. Therefore, most people just created configuration scripts for their projects by copying and modifying the script for a similar project.

In the early 1990s, it was apparent to many open source software developers that project configuration would become painful if something wasn't done to ease the burden of writing massive complex shell scripts to manage configuration options. The number of GNU project packages had grown to hundreds, and maintaining consistency across their separate build systems had become more time-consuming than simply maintaining the code for these projects. These problems had to be solved.

Autoconf

Autoconf[6] changed this paradigm almost overnight. David MacKenzie started the Autoconf project in 1991, but a look at the *AUTHORS* file in the Savannah Autoconf project[7] repository will give you an idea of the number of people who had a hand in making the tool. Although configuration scripts were long and complex, users needed to specify only a few variables when executing them. Most of these variables were simply choices about components, features, and options, such as *Where can the build system find libraries and header files? Where do I want to install my finished products? Which optional components do I want to build into my products?*

6. For more on Autoconf origins, see the GNU web page on the topic at *http://www.gnu.org/software/autoconf/*.

7. See *http://savannah.gnu.org/projects/autoconf/*.

Instead of modifying and debugging hundreds of lines of supposedly portable shell script, developers can now write a short metascript file using a concise, macro-based language, and Autoconf will generate a perfect configuration script that is more portable, more accurate, and more maintainable than a hand-coded one. In addition, Autoconf often catches semantic or logic errors that could otherwise take days to debug. Another benefit of Autoconf is that the shell code it generates is portable between most variations of the Bourne shell. Mistakes made in portability between shells are very common and, unfortunately, are the most difficult kinds of mistakes to find, because no one developer has access to all Bourne-like shells.

NOTE *While portable scripting languages like Perl and Python are now more pervasive than the Bourne shell, this was not the case when the idea for Autoconf was first conceived.*

Autoconf-generated configuration scripts provide a common set of options that are important to all portable software projects running on POSIX systems. These include options to modify standard locations (a concept I'll cover in more detail in Chapter 3), as well as project-specific options defined in the *configure.ac* file (which I'll discuss in Chapter 5).

The autoconf package provides several programs, including the following:

- `autoconf`
- `autoreconf`
- `autoheader`
- `autoscan`
- `autoupdate`
- `ifnames`
- `autom4te`

The `autoconf` program is a simple Bourne shell script. Its main task is to ensure that the current shell contains the functionality necessary to execute the `m4` macro processor. (I'll discuss Autoconf's use of M4 in detail in Chapter 4.) The remainder of the script parses command line parameters and executes `autom4te`.

autoreconf

The `autoreconf` utility executes the configuration tools in the autoconf, automake, and libtool packages as required by the project. This utility minimizes the amount of regeneration required to address changes in timestamps, features, and project state. It was written as an attempt to consolidate existing maintainer-written, script-based utilities that ran all the required Autotools in the right order. You can think of `autoreconf` as a sort of smart Autotools bootstrap utility. If all you have is a *configure.ac* file, you can run `autoreconf` to execute all the tools you need, in the correct order, so that `configure` will be properly generated. Figure 2-1 shows how `autoreconf` interacts with other utilities in the Autotools suite.

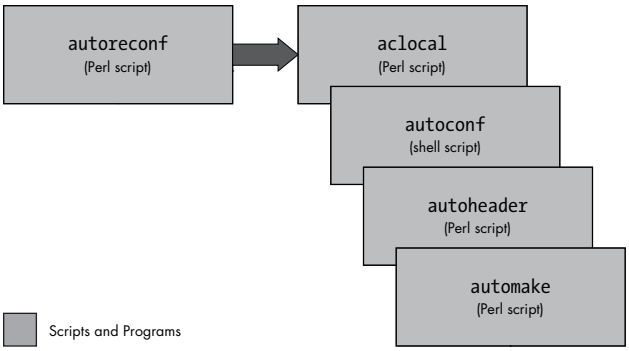

Figure 2-1: A dataflow diagram for the `autoreconf` utility

Nevertheless, there are times when a project requires more than simply bootstrapping the Autotools to get a developer up and running on a newly checked-out repository work area. In these cases, a small shell script that runs `autoreconf`, along with any non-Autotools-related processes, is appropriate. Many projects name such a script `autogen.sh`, but this is often confusing to developers because there is a GNU Autogen project. A better name would be something like `bootstrap.sh`.

Additionally, when used with the `-i` option, `autoreconf` will bootstrap a project into a distributable state by adding missing files that are recommended or required by GNU for proper open source projects. These include a proper *ChangeLog* and template *INSTALL*, *README*, and *AUTHORS* files and so on.

autoheader

The `autoheader` utility generates a C/C++ compatible header file template from various constructs in *configure.ac*. This file is usually called *config.h.in*. When the end user executes `configure`, the configuration script generates *config.h* from *config.h.in*. As maintainer, you'll use `autoheader` to generate the template file you will include in your distribution package. (We'll examine `autoheader` in greater detail in Chapter 4.)

autoscan

The `autoscan` program generates a default *configure.ac* file for a new project; it can also examine an existing Autotools project for flaws and opportunities for enhancement. (We'll discuss `autoscan` in more detail in Chapters 4 and 14.) `autoscan` is very useful as a starting point for a project that uses a non-Autotools-based build system, but it may also be useful for suggesting features that might enhance an existing Autotools-based project.

autoupdate

The `autoupdate` utility is used to update *configure.ac* or the template (*.in*) files to match the syntax supported by current versions of the Autotools.

ifnames

The `ifnames` program is a small and generally underused utility that accepts a list of source file names on the command line and displays a list of C-preprocessor definitions. This utility was designed to help maintainers determine what to put into the *configure.ac* and *Makefile.am* files to make them portable. If your project was written with some level of portability in mind, `ifnames` can help you determine where those attempts at portability are located in your source tree and give you the names of potential portability definitions.

autom4te

The `autom4te` utility is a Perl-based intelligent caching wrapper for `m4` that is used by most of the other Autotools. The `autom4te` cache decreases the time successive tools spend accessing *configure.ac* constructs by as much as 30 percent.

I won't spend a lot of time on `autom4te` (pronounced *automate*) because it's primarily used internally by the Autotools. The only sign that it's working is the *autom4te.cache* directory that appears in your top-level project directory after you run `autoconf` or `autoreconf`.

Working Together

Of the previously listed tools, `autoconf` and `autoheader` are the only ones project maintainers use when generating a `configure` script, and `autoreconf` is the only one that the developer needs to directly execute. Figure 2-2 shows the interaction between input files and `autoconf` and `autoheader` that generates the corresponding product files.

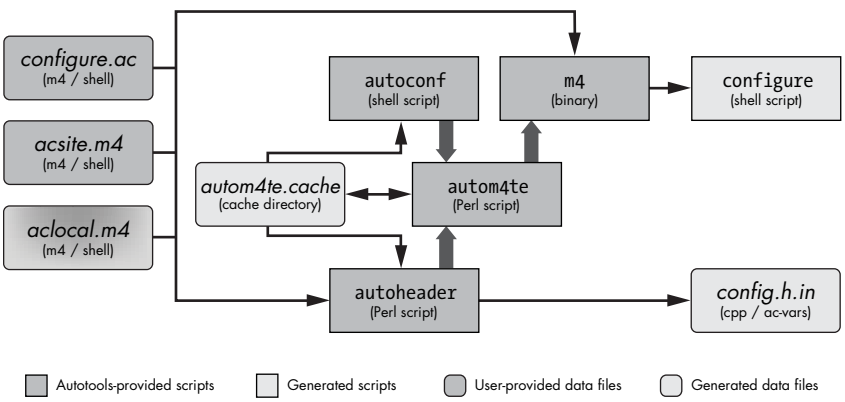

Figure 2-2: A data flow diagram for `autoconf` and `autoheader`

NOTE *I use the data flow diagram format shown in Figure 2-2 throughout this book. Dark boxes represent objects provided either by the user or by an Autotools package. Light boxes represent generated objects. Boxes with square corners are scripts and programs, and boxes with rounded corners are data files. The meaning of most of*

the labels here should be obvious, but at least one deserves an explanation: the term ac-vars *refers to Autoconf-specific replacement text. I'll explain the gradient shading of the* aclocal.m4 *box shortly.*

The primary task of this suite of tools is to generate a configuration script that can be used to configure a project build directory for a target platform (not necessarily the local host). This script does not rely on the Autotools themselves; in fact, `autoconf` is designed to generate configuration scripts that will run on all Unix-like platforms and in most variations of the Bourne shell. This means that you can generate a configuration script using `autoconf` and then successfully execute that script on a machine that does not have the Autotools installed.

The `autoconf` and `autoheader` programs are executed either directly by you or indirectly by `autoreconf`. They take their input from your project's *configure.ac* file and various Autoconf-flavored M4 macro definition files (which, by convention, have a *.m4* extension), using `autom4te` to maintain cache information. The `autoconf` program generates a configuration script called `configure`, a very portable Bourne shell script that enables your project to offer many useful configuration capabilities. The program `autoheader` generates the *config.h.in* template based on certain macro definitions in *configure.ac*.

Automake

Once you've done it a few times, writing a basic makefile for a new project is fairly simple. But problems may occur when you try to do more than just the basics. And let's face it—what project maintainer has ever been satisfied with just a basic makefile?

Attention to detail is what makes an open source project successful. Users lose interest in a project fairly easily—especially when functionality they expect is missing or improperly written. For example, power users have come to expect makefiles to support certain standard targets or goals, specified on the `make` command line, like this:

```
$ make install
```

Common `make` targets include `all`, `clean`, and `install`. In this example, `install` is the target. But you should realize that none of these are *real* targets: a *real target* is a filesystem object that is produced by the build system—usually a file (but sometimes a directory or a link). When building an executable called `doofabble`, for instance, you'd expect to be able to enter:

```
$ make doofabble
```

For this project, `doofabble` is a real target, and this command works for the doofabble project. However, requiring the user to enter real targets on the `make` command line is asking a lot of them, because each project must be built differently—`make doofabble`, `make foodabble`, `make abfooble`, and so on. Standardized targets for `make` allow all projects to be built in the

same way using commonly known commands like `make all` and `make clean`. But *commonly known* doesn't mean *automatic*, and writing and maintaining makefiles that support these targets is tedious and error prone.

Automake's job is to convert a simplified specification of your project's build process into boilerplate makefile syntax that always works correctly the first time *and provides all the standard functionality expected*. Automake creates projects that support the guidelines defined in the *GNU Coding Standards* (discussed in Chapter 3).

Just like `autoconf` produces a `configure` script that is portable to many flavors of the Bourne shell, `automake` produces `make` script that is portable to many flavors of `make`.

The automake package provides the following tools in the form of Perl scripts:

- `automake`
- `aclocal`

automake

The `automake` program generates standard makefile templates (named *Makefile.in*) from high-level build specification files (named *Makefile.am*). These *Makefile.am* input files are essentially just regular makefiles. If you were to put only the few required Automake definitions in a *Makefile.am* file, you'd get a *Makefile.in* file containing several hundred lines of parameterized `make` script.

If you add additional `make` syntax to a *Makefile.am* file, Automake will move this code to the most functionally correct location in the resulting *Makefile.in* file. In fact, you can write your *Makefile.am* files so all they contain is ordinary `make` script, and the resulting makefiles will work just fine. This pass-through feature gives you the ability to extend Automake's functionality to suit your project's specific requirements.[8]

aclocal

In the *GNU Automake Manual*, the `aclocal` utility is documented as a temporary workaround for a certain lack of flexibility in Autoconf. Automake enhances Autoconf by adding an extensive set of macros, but Autoconf was not really designed with this level of enhancement in mind.

The original documented method for adding user-defined macros to an Autoconf project was to create a file called *aclocal.m4*, place the user-defined macros in this file, and place the file in the same directory as *configure.ac*. Autoconf then automatically includes this set of macros while processing *configure.ac*. The designers of Automake found this extension mechanism too useful to pass up; however, users would have been required to add an `m4_include` statement to a possibly unnecessary *aclocal.m4* file in order to

8. Other metabuild tools like CMake also generate makefiles but do not allow you to directly specify what ends up in these files. Rather, you have to find the correct approach in CMake's macro language in order to coerce it into writing `make` script that does what you want it to.

include the Automake macros. Since both user-defined macros and the use of M4 itself are considered advanced concepts, this was deemed too harsh a requirement.

The `aclocal` script was designed to solve this problem. This utility generates an *aclocal.m4* file for a project that contains both user-defined macros and all required Automake macros.[9] Instead of adding user-defined macros directly to *aclocal.m4*, project maintainers should now add them to a new file called *acinclude.m4*.

To make it clear to readers that Autoconf doesn't depend on Automake (and perhaps due to a bit of stubbornness), the *GNU Autoconf Manual* doesn't make much mention of the `aclocal` utility. The *GNU Automake Manual* originally suggested that you rename *aclocal.m4* to *acinclude.m4* when adding Automake to an existing Autoconf project, and this approach is still commonly used. The flow of data for `aclocal` is depicted in Figure 2-3.

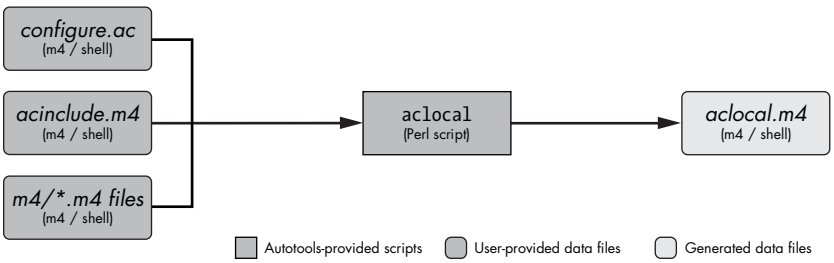

Figure 2-3: A data flow diagram for `aclocal`

However, the latest documentation for both Autoconf and Automake suggests that the entire paradigm is now obsolete. Developers should now specify a directory that contains a set of M4 macro files. The current recommendation is to create a directory in the project root directory called *m4* and add macros as individual *.m4* files to it. All files in this directory will be gathered into *aclocal.m4* before Autoconf processes *configure.ac*.[10]

It may now be more apparent why the *aclocal.m4* box in Figure 2-2 couldn't decide which color it should be. When you're using it without Automake and Libtool, you write *aclocal.m4* by hand. However, when you're using it with Automake, the file is generated by the `aclocal` utility, and you provide project-specific macros either in *acinclude.m4* or in an *m4* directory.

Libtool

How do you build shared libraries on different Unix platforms without adding a lot of very platform-specific conditional code to your build system and source code? This is the question that the Libtool project tries to address.

9. Automake macros are copied into this file, but the user-written *acinclude.m4* file is merely referenced with an `m4_include` statement at the end of the file.

10. As with *acinclude.m4*, this gathering is virtual; *aclocal.m4* merely contains `m4_include` statements that reference these other files in place.

There's a significant amount of common functionality among Unix-like platforms. However, one very significant difference has to do with how shared libraries are built, named, and managed. Some platforms name their libraries *lib*name.*so*, others use *lib*name.*a* or even *lib*name.*sl*. The Cygwin system for Windows names Cygwin-generated shared libraries *cyg*name.*dll*. Still others don't even provide native shared libraries. Some platforms provide *libdl.so* to allow software to dynamically load and access library functionality at runtime, while others provide different mechanisms, and some platforms don't provide this functionality at all.

The developers of Libtool have carefully considered all of these differences. Libtool supports dozens of platforms, not only providing a set of Autoconf macros that hide library-naming differences in makefiles but also offering an optional library of dynamic loader functionality that can be added to programs. This functionality allows maintainers to make their runtime, dynamic shared-object management code more portable and easier to maintain.

The libtool package provides the following programs, libraries, and header file:

- `libtool` (program)
- `libtoolize` (program)
- *ltdl* (static and shared libraries)
- *ltdl.h* (header file)

libtool

The `libtool` shell script that ships with the libtool package is a generic version of the custom script that `libtoolize` generates for a project.

libtoolize

The `libtoolize` shell script prepares your project to use Libtool. It generates a custom version of the generic `libtool` script and adds it to your project directory. This custom script is shipped with the project along with the Automake-generated makefiles, which execute the script on the user's system at the appropriate time.

ltdl, the Libtool C API

The libtool package also provides the *ltdl* library and associated header files, which provide a consistent runtime shared-object manager across platforms. The *ltdl* library may be linked statically or dynamically into your programs, giving them a consistent runtime shared-library access interface between platforms.

Figure 2-4 illustrates the interaction between the `automake` and `libtool` scripts, and the input files used to create products that configure and build your projects.

Automake and Libtool are both standard pluggable options that can be added to *configure.ac* with just a few simple macro calls.

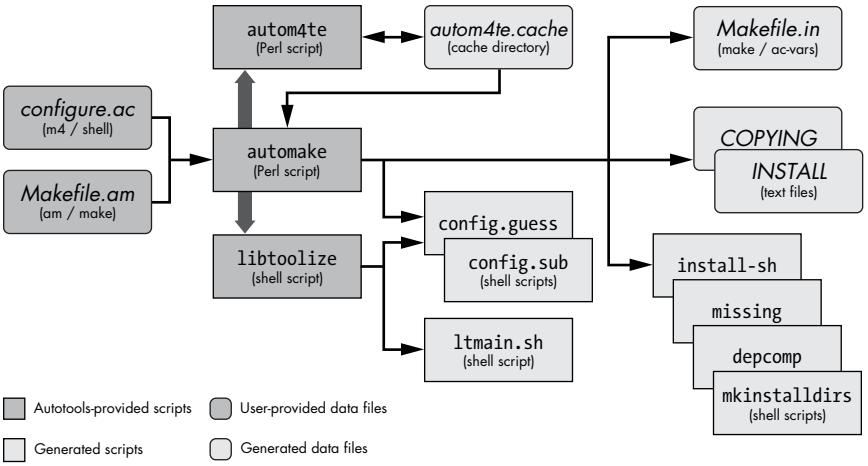

Figure 2-4: A data flow diagram for `automake` and `libtool`

Building Your Package

As maintainer, you probably build your software packages fairly often, and you're also probably intimately familiar with your project's components, architecture, and build system. However, you should make sure that your users' build experiences are much simpler than your own. One way to do this is to give users a simple, easy-to-understand pattern to follow when building your software packages. In the following sections, I'll show you the build pattern supported by the Autotools.

Running configure

After running the Autotools, you're left with a shell script called `configure` and one or more *Makefile.in* files. These files are intended to be shipped with your project release distribution packages.[11] Your users will download these packages, unpack them, and enter `./configure && make` from the top-level project directory. The `configure` script will generate makefiles (called *Makefile*) from the *Makefile.in* templates created by `automake` and a *config.h* header file from the *config.h.in* template generated by `autoheader`.

Automake generates *Makefile.in* templates rather than makefiles because without makefiles, your users can't run `make`; you don't want them to run `make` until after they've run `configure`, and this functionality guards against them doing so. *Makefile.in* templates are nearly identical to makefiles you might

11. GPL licensing also requires *configure.ac* and *Makefile.am* to be shipped with your package, and the Autotools ensure that these files are in the distribution tarball. The reasoning is that the GPL requires the full source of a project to be distributed in preferred-editing form. A user obtaining the distribution tarball would not be able to edit anything without the base source files for the build system. However, end users need not touch or interact with these files unless they wish to customize the program in a manner not supported by project configuration options.

write by hand, except that you didn't have to. They also do a lot more than most people are willing to hand-code. Another reason for not shipping ready-to-run makefiles is that it gives `configure` the chance to insert platform characteristics and user-specified optional features directly into the makefiles. This makes them a better fit for their target platforms and the end user's build preferences. Finally, the makefiles can also be generated outside the source tree, which means you can create custom build systems in different directories for the same source directory tree. I'll discuss this topic in greater detail in "Building Outside the Source Directory" on page 28.

Figure 2-5 illustrates the interaction between `configure` and the scripts it executes during the configuration process in order to create the makefiles and the *config.h* header file.

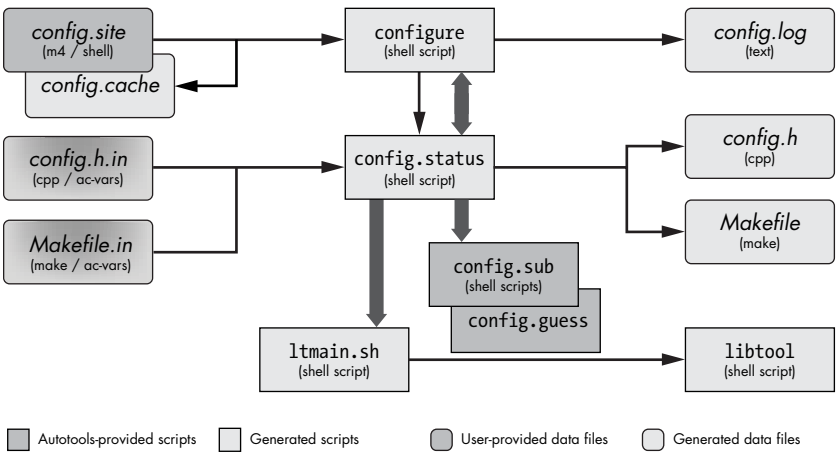

Figure 2-5: A data flow diagram for `configure`

The `configure` script has a bidirectional relationship with another script called `config.status`. You may have thought that your `configure` script generated your makefiles. But actually, the only file (besides a log file) that `configure` generates is `config.status`.

The `configure` script is designed to determine platform characteristics and features available on the user's system, as specified in the maintainer-written *configure.ac*. Once it has this information, it generates `config.status`, which contains all of the check results, and then it executes this script. The `config.status` script, in turn, uses the check information embedded within it to generate platform-specific *config.h* and makefiles, as well as any other template-based output files specified in *configure.ac*.

NOTE *As the double-ended fat arrow in Figure 2-5 shows, `config.status` can also call `configure`. When used with the `--recheck` option, `config.status` will call `configure` using the same command line options used to originally generate `config.status`.*

The `configure` script also generates a log file called *config.log*, which will contain very useful information in the event that an execution of `configure` fails on the user's system. As the maintainer, you can use this information

for debugging. The *config.log* file also logs how `configure` was executed. (You can run `config.status --version` to discover the command line options used to generate `config.status`.) This feature can be particularly handy when, for example, a user returns from vacation and can't remember which options they used to originally generate the project build directory.

NOTE *To regenerate makefiles and the* config.h *header files, just enter* `./config.status` *from within the project build directory. The output files will be generated using the same options originally used to generate* `config.status`.

The *config.site* file can be used to customize the way `configure` works based on the `--prefix` option passed to it. The *config.site* file is a script, but it's not meant to be executed directly. Rather, `configure` looks for *$(prefix)/share/config.site* and "sources" it (incorporates it as part of its own script) before executing any of its own code. This can be a handy way of specifying the same set of options for many packages, all destined to be built and installed the same way. Since `configure` is just a shell script, *config.site* should just contain shell code.

The *config.cache* file is generated by `configure` when the `-C` or `--config-cache` options are used. The results of configuration tests are cached in this file and are reusable by subdirectory `configure` scripts or by future runs of `configure`. By default, *config.cache* is disabled because it can be a potential source of configuration errors. If you're confident with your configuration process, *config.cache* can really speed up the configuration process between executions of `configure`.

Building Outside the Source Directory

A little-known feature of Autotools build environments is that they don't need to be generated within a project source tree. That is, if a user executes `configure` from a directory other than the project source directory, they can generate a full build environment within an isolated build directory.

In the following example, the user downloads *doofabble-3.0.tar.gz*, unpacks it, and creates two sibling directories called *doofabble-3.0.debug* and *doofabble-3.0.release*. They change into the *doofabble-3.0.debug* directory; execute doofabble's `configure` script, using a relative path, with a doofabble-specific `debug` option; and then run `make` from within this same directory. Then they switch over to the *doofabble-3.0.release* directory and do the same thing, this time running `configure` without the `debug` option:

```
$ gzip -dc doofabble-3.0.tar.gz | tar xf -
$ mkdir doofabble-3.0.debug
$ mkdir doofabble-3.0.release
$ cd doofabble-3.0.debug
$ ../doofabble-3.0/configure --enable-debug
--snip--
$ make
--snip--
$ cd ../doofabble-3.0.release
$ ../doofabble-3.0/configure
```

```
--snip--
$ make
--snip--
```

Users generally don't care about remote build functionality, because all they usually want to do is configure, build, and install your code on their platforms. Maintainers, on the other hand, find remote build functionality very useful, as it allows them to not only maintain a reasonably pristine source tree but also to maintain multiple build environments for their project, each with complex configuration options. Rather than reconfigure a single build environment, a maintainer can simply switch to another build directory that has been configured with different options.

There is one case, however, where a user might wish to use remote-build. Consider the case where one obtains the full unpacked source code of a project on CD or has access to it via a read-only NFS mount. The ability to build outside the source tree can grant the ability to build the project without having to copy it to writable media.

Running make

Finally, you run plain old `make`. The designers of the Autotools went to a *lot* of trouble to ensure that you didn't need any special version or brand of `make`. Figure 2-6 depicts the interaction between `make` and the makefiles that are generated during the build process.

NOTE *There has been some discussion on the Autotools mailing lists during the last few years about supporting only GNU `make`, as modern GNU `make` is so much more functional than other `make` utilities. Almost all Unix-y platforms (and even Microsoft Windows) have a version of GNU `make` today, so the rationale for continuing to support other brands of `make` is no longer as important as it once was.*

As you can see, `make` runs several generated scripts, but these are all really ancillary to the `make` process. The generated makefiles contain commands that execute these scripts under the appropriate conditions. These scripts are part of the Autotools, and they are either shipped with your package or generated by your configuration script.

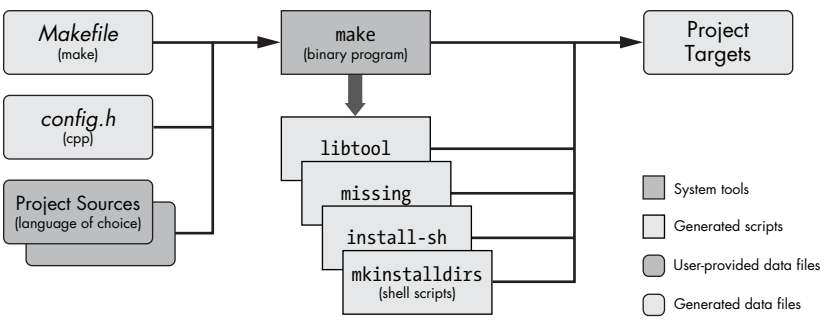

Figure 2-6: A data flow diagram for `make`

Installing the Most Up-to-Date Autotools

If you're running a variant of Linux and you've chosen to install the compilers and tools used for developing C-language software, you probably already have some version of the Autotools installed on your system. To determine which versions of Autoconf, Automake, and Libtool you're using, simply open a terminal window and type the following commands (if you don't have the `which` utility on your system, try **`type -p`** instead):

```
$ which autoconf
/usr/local/bin/autoconf
$
$ autoconf --version
autoconf (GNU Autoconf) 2.69
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+/Autoconf: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>, <http://gnu.org/licenses/exceptions.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by David J. MacKenzie and Akim Demaille.
$
$ which automake
/usr/local/bin/automake
$
$ automake --version
automake (GNU automake) 1.15
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv2+: GNU GPL version 2 or later <http://gnu.org/licenses/gpl-
2.0.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Tom Tromey <tromey@redhat.com>
       and Alexandre Duret-Lutz <adl@gnu.org>.
$
$ which libtool
/usr/local/bin/libtool
$
$ libtool --version

libtool (GNU libtool) 2.4.6
Written by Gordon Matzigkeit, 1996

Copyright (C) 2014 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.
$
```

NOTE *If you have the Linux-distribution varieties of these Autotools packages installed on your system, the executables will probably be found in* /usr/bin *rather than* /usr/local/bin, *as you can see from the output of the* `which` *command here.*

If you choose to download, build, and install the latest released version of any one of these packages from the GNU website, you must do the same for all of them, because the Automake and Libtool packages install macros into the Autoconf macro directory. If you don't already have the Autotools installed, you can install them using your system package manager (for example, `yum` or `apt`), or from source, using their GNU distribution source archives. The latter can be done with the following commands (be sure to change the version numbers as necessary):

```
$ mkdir autotools && cd autotools
$ wget -q https://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz
$ wget -q https://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz.sig
$ gpg autoconf-2.69.tar.gz.sig
gpg: assuming signed data in `autoconf-2.69.tar.gz'
gpg: Signature made Tue 24 Apr 2012 09:17:04 PM MDT using RSA key ID 2527436A
gpg: Can't check signature: public key not found
$
$ gpg --keyserver keys.gnupg.net --recv-key 2527436A
gpg: requesting key 2527436A from hkp server keys.gnupg.net
gpg: key 2527436A: public key "Eric Blake <eblake@redhat.com>" imported
gpg: key 2527436A: public key "Eric Blake <eblake@redhat.com>" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 2
gpg:               imported: 2  (RSA: 2)$ gpg autoconf-2.69.tar.gz.sig
gpg: assuming signed data in `autoconf-2.69.tar.gz'
gpg: Signature made Tue 24 Apr 2012 09:17:04 PM MDT using RSA key ID 2527436A
gpg: Good signature from "Eric Blake <eblake@redhat.com>"
gpg:                 aka "Eric Blake (Free Software Programmer) <ebb9@byu.net>"
gpg:                 aka "[jpeg image of size 6874]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 71C2 CC22 B1C4 6029 27D2  F3AA A7A1 6B4A 2527 436A
$
$ gzip -cd autoconf* | tar xf -
$ cd autoconf*/
$ ./configure && make all check
       # note - a few tests (501 and 503, for example) may fail
       # - this is fine for this release)
--snip--
$ sudo make install
--snip--
$ cd ..
$ wget -q https://ftp.gnu.org/gnu/automake/automake-1.16.1.tar.gz
$ wget -q https://ftp.gnu.org/gnu/automake/automake-1.16.1.tar.gz.sig
$ gpg automake-1.16.1.tar.gz.sig
gpg: assuming signed data in `automake-1.16.1.tar.gz'
gpg: Signature made Sun 11 Mar 2018 04:12:47 PM MDT using RSA key ID 94604D37
gpg: Can't check signature: public key not found
$
$ gpg --keyserver keys.gnupg.net --recv-key 94604D37
gpg: requesting key 94604D37 from hkp server keys.gnupg.net
gpg: key 94604D37: public key "Mathieu Lirzin <mthl@gnu.org>" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
```

```
gpg:               imported: 1  (RSA: 1)
$
$ gpg automake-1.16.1.tar.gz.sig
gpg: assuming signed data in `automake-1.16.1.tar.gz'
gpg: Signature made Sun 11 Mar 2018 04:12:47 PM MDT using RSA key ID 94604D37
gpg: Good signature from "Mathieu Lirzin <mthl@gnu.org>"
gpg:                 aka "Mathieu Lirzin <mthl@openmailbox.org>"
gpg:                 aka "Mathieu Lirzin <mathieu.lirzin@openmailbox.org>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: F2A3 8D7E EB2B 6640 5761  070D 0ADE E100 9460 4D37
$
$ gzip -cd automake* | tar xf -
$ cd automake*/
$ ./configure && make all check
--snip--
$ sudo make install
--snip--
$ cd ..
$ wget -q https://ftp.gnu.org/gnu/libtool/libtool-2.4.6.tar.gz
$ wget -q https://ftp.gnu.org/gnu/libtool/libtool-2.4.6.tar.gz.sig
$ gpg libtool-2.4.6.tar.gz.sig
gpg: assuming signed data in `libtool-2.4.6.tar.gz'
gpg: Signature made Sun 15 Feb 2015 01:31:09 PM MST using DSA key ID 2983D606
gpg: Can't check signature: public key not found
$
$ gpg --keyserver keys.gnupg.net --recv-key 2983D606
gpg: requesting key 2983D606 from hkp server keys.gnupg.net
gpg: key 2983D606: public key "Gary Vaughan (Free Software Developer) <gary@vaughan.pe>"
imported
gpg: key 2983D606: public key "Gary Vaughan (Free Software Developer) <gary@vaughan.pe>"
imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 2
gpg:               imported: 2  (RSA: 1)
$
$ gpg libtool-2.4.6.tar.gz.sig
gpg: assuming signed data in `libtool-2.4.6.tar.gz'
gpg: Signature made Sun 15 Feb 2015 01:31:09 PM MST using DSA key ID 2983D606
gpg: Good signature from "Gary Vaughan (Free Software Developer) <gary@vaughan.pe>"
gpg:                 aka "Gary V. Vaughan <gary@gnu.org>"
gpg:                 aka "[jpeg image of size 9845]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: CFE2 BE70 7B53 8E8B 2675  7D84 1513 0809 2983 D606
$
$ gzip -cd libtool* | tar xf -
$ cd libtool*/
$ ./configure && make all check
--snip--
$ sudo make install
--snip--
$ cd ..
$
```

The preceding example shows how to use the associated *.sig* files to validate the signature on GNU packages. The example assumes you have not configured a gpg key server on your system and that you have not installed the public key for any of these packages. If you have already configured a preferred key server, you can skip the `gpg` command line `--keyserver` options. Once you've imported the public keys for these packages, you need not do it again.

You may also wish to install in a manner that does not require root access via `sudo`. To do this, execute `configure` with a `--prefix` option such as `--prefix=$HOME/autotools` and then add *~/autotools/bin* to your `PATH` environment variable.

You should now be able to successfully execute the version-check commands from the previous example. If you still see older versions, ensure your `PATH` environment variable properly contains */usr/local/bin* (or wherever you installed to) before */usr/bin*.

Summary

In this chapter, I presented a high-level overview of the Autotools to give you a feel for how everything ties together. I also showed you the pattern to follow when building software from distribution tarballs created by Autotools build systems. Finally, I showed you how to install the Autotools and how to tell which versions you have installed.

In Chapter 3, we'll step away from the Autotools briefly and begin creating a hand-coded build system for a toy project called *Jupiter*. You'll learn the requirements of a reasonable build system, and you'll become familiar with the rationale behind the original design of the Autotools. With this background knowledge, you'll begin to understand why the Autotools do things the way they do. I can't really emphasize this enough: *Chapter 3 is one of the most important chapters in this book, because it will get you past any emotional stigma you may have associated with the Autotools due to misconceptions.*