Autotools: a practitioner's guide to Autoconf, Automake and Libtool

There are few people who would deny that Autoconf, Automake and Libtool have revolutionized the free software world. While there are many thousands of Autotools advocates, some developers absolutely *hate* the Autotools, with a passion. Why? Let me try to explain with an analogy.
In the early 1990's I was working on the final stages of my bachelor's degree in computer science at Brigham Young University. I took a 400-level computer graphics class, wherein I was introduced to C++, and the object-oriented programming paradigm. For the next 5 years, I had a love-hate relationship with C++. I was a pretty good C coder by that time, and I thought I could easily pick up C++, as close in syntax as it was to C. How wrong I was. I fought late into the night, more often than I'd care to recall, with the C++ compiler over performance issues.

The problem was that the most fundamental differences between C++ and C are not obvious to the casual observer. Most of these differences are buried deep within the C++ language specification, rather than on the surface, in the language syntax. The C++ compiler generates code beneath the covers at a level never even conceived of by C compiler writers. This level of code generation provides functionality in a few lines of C++ code that requires dozens of lines of C code. Oh, yes--you can write object-oriented software in C. But you are required to manage all of the details yourself. In C++, these details are taken care of for you by the compiler. The advantages should be clear.

But this high-level functionality comes at a price--you have to learn to understand what the compiler is doing for you, so you can write your code in a way that complements it. Not surprisingly, often the most intuitive thing to do in this situation for the new C++ programmer is to inadvertently write code that works against the underlying infrastructure generated by the compiler.

And therein lies the problem. Just as there were many programmers then (I won't call them software engineers--that title comes with experience, not from a college degree) complaining of the nightmare that was C++, so likewise there are many programmers today complaining of the nightmare that is the Autotools. The differences between make and Automake are very similar to the differences between C and C++. The most basic single-line Makefile.am generates a Makefile.in file (an Autoconf template) containing nearly 350 lines of make script.

Who should read this book

This book is written for the open source software package maintainer. I'm purposely not using the terms "free software" or "proprietary software that's free". The use of the term "open source" is critical in this context. You see, open source defines a type of software distribution channel. One in which the primary method of obtaining software functionality is downloading a source archive, unpacking, building and installing the built products on your system. Free software may be published in binary form. Proprietary software may be given away. But open source software implies source-level distribution.

Source-level distribution relegates a particular portion of the responsibility of software development to the end-user that has traditionally been assumed by the software developer. But end-users are not developers, so most of them won't know how to properly build your package. What to do, what to do... The most widely adopted approach

Chapter 1: A brief introduction to the GNU Autotools

Mon, 2008-05-12 02:51 -- John Calcote

I'm going to make a rather broad and sweeping statement here: If you're writing free or open source software targeting Unix or Linux systems, then you should be using the GNU Autotools. I'm sure I sound a bit biased, but I'm not. And I shouldn't be, given the number of long nights I've spent working around what appeared to be shortcomings in the Autotools system. Normally, I would have been angry enough to toss the entire project out the window and write a good hand-coded makefile and configure script. But the one fact that I always came back to was that there are literally thousands of projects out there that appear to be very successfully using the Autotools. This was too much for me. My pride would never let me give up.

Who should use the Autotools?

The Autotools are supposed to make projects simpler for the maintainer, right? And the answer to that question is a definitive "No". Don't misunderstand me here--the Autotools do make your life easier in the long run, but for different reasons than you may first realize. The primary goal of the Autotools is not to make project maintenance simpler, although I honestly believe the system is as simple as it can be, given the functionality it provides. It took me a while to figure this out, and really, it was one of my most significant Autotools epiphanies. Ultimately, I came to understand that the purpose of the Autotools is two-fold: First, to make life easer for your *users*, and second, to make your project more portable--even to systems on which you've never tested, installed or even built your code.

Well then, what if you don't work on free or open source software? Do you still care about these goals? What if you're writing proprietary software for Unix or Linux systems? Then, I say, you would probably still benefit to some degree from using the Autotools. Even if you only ever intend to target a single distribution of Linux, the Autotools will provide you with a build environment that is flexible enough to allow your project to build successfully on future versions or distributions with virtually no changes to the build scripts. And, let's be honest here--you really *can't* know in advance whether or not your management will want your software to run on other platforms in the future. This fact alone is enough to warrant my statement.

Who should NOT use the Autotools?

About the only scenario where it makes sense NOT to use the Autotools is the one in which you are writing software for non-Unix platforms only--Microsoft Window comes to mind. Some people will tell you that the Autotools can be used successfully on Windows as well, but my opinion is that the POSIX/FHS approach to software build management is just too alien for Windows development. While it *can* be done, the tradeoffs are way too significant to justify shoe-horning a Windows project into the Autotools build paradigm. I've watched some project managers develop custom versions of the Autotools which allow the use of all native Windows tools. These projects were maintained by people who spent much of their time tweaking the tools and the build environment to do things it was never intended to do, in a hostile and foreign environment. Quite frankly, Microsoft has some of the best tools on the planet for Windows software development. If I were developing a Windows software package, I'd use Microsoft's tools exclusively. In fact, I often write portable software that targets both Linux and Windows. In these cases, I maintain two separate build environments--one for Windows, and one based on the Autotools for everything else.

The original reasons for using GNU tools to build Windows software were that GNU tools were free, and Microsoft tools were expensive. This reason is no longer valid, as Microsoft makes the better part of their tools available for free download today. This was a smart move on their part--but it took them long enough to see the value in it.

Your choice of language

One other important factor in the decision to use or not use the Autotools with your project is your language of choice. Let's face it, the Autotools were designed by GNU people to manage GNU projects. There are two factors that determine the importance of a computer language within the GNU community:

- 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:

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

By "native support", I mean that Autoconf will compile, link and run source-level feature checks in these languages.

If you want to build a Java package, you can configure Automake to do so, but you can't ask Autoconf to compile, link or run Java-based checks. Java simply isn't supported natively at this time by Autoconf. I believe it's important to point out here that the very nature of the Java language and virtual machine specifications make it far less likely that you'll need to perform a Java-based Autoconf check in the first place.

There is work being actively done on the gcj compiler and tool set, so it's not unreasonable to think that some native Java support will be added to Autoconf at some future date, but gcj is a bit immature yet, and currently very few (if any) GNU packages are written in Java, so the issue is not critical to the GNU community.

That said, there is currently rudimentary support in Automake for both GNU (gcj) and non-GNU Java compilers and VM's. I've used it myself on a project, and it works well, as long as you don't try to push it too far. Given the history of the GNU project, I think it's safe to say that this functionality will definitely improve with age.

If you're into Smalltalk, ADA, Modula, LISP, Forth, or some other non-mainstream language, well then you're probably not too concerned about porting your code to dozens of platforms and CPUs.

As an aside, if you *are* using a non-mainstream language, and you are in fact concerned about the portability of your build systems, then please consider adding support for your language to the Autotools. This is not as daunting a task as you may think, and I gaurantee that you'll be an Autotools expert when you're finished. If you think this statement is funny, then consider how Erlang support made it into the Autotools. I'm betting most developers have never heard of Erlang, but members of the Erlang community thought it was important enough to add Erlang support themselves.

Generating your package build system

The GNU Autotools framework is comprised of three main packages, each of which provides and relies on several smaller components. The three main packages are Autoconf, Automake and Libtool. These packages were invented in that order, and evolved over time. Additionally, the tools in the Autotools packages can depend on or use utilities and functionality from the gettext, m4, sed, make and perl packages, as well as others.

It's very important at this point 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 rely only on readily available, preinstalled tools on the host machine. Perl is only required on machines that maintainers use to create distributions, not on end-user machines that build packages from resulting release distributions packages. A corollary to this is that end-users' machines need not have the Autotools installed.

If you've ever downloaded, built and installed software from a "tarball"--a compressed archive with a .tar.gz, .tgz or .tar.bz2 extension--then you're probably aware of the fact that there is a common theme to this process. It usually looks something like this:

```
$ gzip -cd hackers-delight-1.0.tar.gz | tar -xvf -

...

$ cd hackers-delight-1.0

$ ./configure

$ make all

$ sudo make install
```

NOTE: I have to assume some level of knowledge on your part, and I'm stating right now that this is it. If you've performed this sequence of commands before and you know what it means, and if you have a basic understanding of the software development process, then you'll have no trouble following the content of this book.

Most developers know and understand the purpose of the `make` utility. But what's the point of the configure script? The use of configuration scripts (generally named `configure`) started a long time ago on Unix systems due to the variety imposed by the fast growing and divergent set of Unix and Unix-like platforms. It's interesting to note that while Unix systems have generally followed the defacto-standard Unix kernel interface for decades, most software that does anything significant generally has to stretch outside of these more or less standardized boundaries. Configuration scripts are hand-coded shell scripts designed to determine platform-specific characteristics, and to allow users to choose package options before running make.

This approach worked well for decades. With the advent of dozens of Linux distributions, the explosion of feature permutations has made writing a decent portable configuration script very difficult--much more so than writing the makefiles for a new project. Most people have come up with configuration scripts for their projects using a well-understood and pervasive technique--copy and modify a similar project's script. By the early 90's it was becoming apparent to many developers that project configuration was going to become painful if something weren't done to ease the burden of writing massive shell scripts to manage configuration options--both those related to platform differences, and those related to package options.

Autoconf

Autoconf changed this paradigm almost overnight. A quick glance at the AUTHORS file in the Savannah Autoconf project repository will give you an idea of the number of people that have had a hand in the making of Autoconf. The original author was David MacKenzie, who started the Autoconf project in 1991. While configuration scripts were becoming longer and more complex, there were really only a few variables that needed to be specified by the user. Most of these were simply choices to be made regarding components, features and options: Where do I find libraries and header files? Where do I want to install my finished product? Which optional components do I want to build into my products? With Autoconf, instead of modifying, debugging and losing sleep over literally thousands of lines of supposedly portable shell script, developers can write a short meta-script file, using a concise macro-based language, and let Autoconf generate a perfect configuration script.
A generated configuration script is more portable, more correct, and more maintainable than a hand-code version of the same script. In addition, Autoconf often catches semantic or logic errors that the author would have spent days debugging. Another benefit of Autoconf is that the shell code it generates is as portable as possible between systems that supply any form of the Bourne shell. Mistakes made in portability between shells are by far the most common, and unfortunately the most difficult to find, because no one programmer has access to all versions or brands of Bourne-like shells in existence.

Autoconf generated configure scripts provide a common set of options that are important to all portable, free, open source, and proprietary software projects running on LSB-compliant systems. These include options to modify "standard locations", a concept I'll cover in more detail in Chapter 2. Autoconf generated configure scripts also provide project-specific options. These are defined in the configure.ac file for each project. I'll detail this process in Chapter 3.

The Autoconf package provides several programs. Autoconf itself is written in Bourne shell script, while the others are perl scripts.

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

Autoheader

The autoheader utility generates a C language header file *template* from configure.ac. This template file is usually called config.h.in. We'll cover autoheader in greater detail in Chapter 3.

Autom4te

The autom4te utility is a cache manager used by most of the other Autotools. In the early days of Autoconf there was really no need for such a cache, but because most of the Autotools use constructs found in configure.ac, the cache speeds up access by successive programs to configure.ac by about 40 percent or more. I won't spend a lot of time on autom4te (which is pronounced "automate", by the way), because it's mainly used internally by the Autotools, and the only sign you're given that it's working is the existence of an autom4te.cache directory in your top-level project directory.

Autoreconf

The autoreconf program can be used to execute the configuration tools in the Autoconf, Automake and Libtool packages as required by the project. The purpose of autoreconf is to minimize the amount of regeneration that needs to be done, based on timestamps, features, and project state. Think of autoreconf as an Autotools bootstrap utility. If all you have is a configure.ac file, running autoreconf will run the tools you need in order to run configure and then make.

Autoscan

The autoscan program is used to generate a reasonable configure.ac file for a new project. We'll spend some time on autoscan later in Chapter's 3 and 6, as we go through the process of setting up the Autotools on a basic project.

Autoupdate

The autoupdate utility is used to update your configure.ac or template (*.in) files to the syntax of the current version of the Autotools. I'll cover autoupdate in more detail in Chapter 2.

Ifnames

The ifnames program is a small, and generally under-utilized program that accepts a list of source files names on the command line, and displays a list of C preprocessor definitions and their containing files on the stdout device. This utility is designed to help you determine what to put into your configure.ac and Makefile.am files for the sake of portability. If your project has already been written with some level of portability in mind, ifnames can help you find out where those attempts are located in your source tree, and what the names of potential portability defintions might be.

Of the tools in this list, only autoconf and autoheader are used directly by the project maintainer while generating a configure script, and actually, as we'll see later, only autoreconf really needs to be called directly. The following diagram shows the interaction between input files and the Autoconf and autoheader programs to generate product files:


Figure 1: Autoconf and autoheader data flow diagram

NOTE: I'll follow this data flow diagram format through the rest of this book. Darker colored boxes represent objects that are provided either by the user or by an Autotools package. Lighter shades of the same colors represent generated objects of the same type.

These tools' primary task is to generate a configure script that can be used by you or others to configure a project build directory. The configure script generated does not rely in any way on the Autotools themselves. In fact, Autoconf is *specifically designed* to generate configure scripts that will run on all Unix or Unix-like platforms that support a Bourne shell. You should be able to generate a configure script from Autoconf, and then successfully execute that script on a machine which does not have the Autotools installed. Not surprisingly, this is actually a common use-case in the free software world, so it's also a well-tested use case.

As you can see in this diagram, Autoconf and autoheader are called by the user. These tools take their input from your project's configure.ac file, and various Autoconf-flavored m4 macro definition files. They use autom4te to maintain cache information. Autoconf generates your configure script, a very portable Bourne shell script that provides your project with configuration capabilities. Autoheader generates the config.h.in template based on macro definitions in configure.ac.

You may have noticed the apparent identity crisis being suffered by the aclocal.m4 input file. Is that a bit of a blush on that box--is it a generated file, or a user-provided file? Well, the answer is that it's both, and I'll explain this in more detail in the next section.

Automake

So, what's so difficult about writing a makefile? Well, actually, once you've done it a few times, writing a *basic* makefile for a new project is really rather trivial. The problems 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?
The single most significant difference between a successful free software project and one that rarely gets a second glance can be found deep in the heart of project maintenance details. These details include providing the so-called "standard make targets". Potential users become disgusted with a project fairly easily--especially when certain bits of expected functionality are missing or improperly written. Users have come to expect certain more or less standard make targets. A make target is a goal specified on the make command line:

```
$ make install
```

In this example, `install` is the goal or target. Common make targets include `all`, `clean` and `install`, among others. You'll note that none of these are *real* targets. A real target is a file produced by the build system. If you're building an executable called doofabble, then you'd expect to be able to type:

```
$ make doofabble
```

This would generate an actual executable file called `doofabble`. But specifying real targets on the make command line is more work than necessary. Each project must be built differently--make doofabble, make foodabble, make abfooble, etc. Why not just type make or make all, if there is more than one binary to be made? So `all` has become an expected pseudo-target, but "expected" doesn't mean "automatic".
Supporting the expected set of standard targets can be fairly challenging. As with configuration scripts, the most widely used implementation is one written in the late 80's and copied from project to project throughout the internet. Why? Because writing it yourself is error prone. In fact, copying it is just as error-prone. It's like getting a linked-list implementation right the first time. The process is well-understood by any veteran software engineer, but it still rarely happens. Object-oriented programming languages like C++ and Java provide libraries and templates for these constructs now--not because they're hard to implement by hand, but because doing so is error-prone, and there's no point in re-inventing the wheel--yet again.

Automake's job is to convert a much simplified specification of your project's build process into standard boilerplate makefile syntax that always works correctly the first time, and provides all the standard functionality expected of a free software project. In actuality, Automake creates projects that support guidelines defined in the GNU Coding Standards, which I'll cover in greater detail in Chapter 2.

The Automake package provides the following tools in the form of perl scripts:

- automake
- aclocal

The primary task of the Automake program is to generate standard makefile templates (named Makefile.in) from high-level build specification files (named Makefile.am). One of the most interesting and useful aspects of the way Automake works is that the Makefile.am input files are mostly just regular makefiles. If you put only the few required Automake definitions in a Makefile.am, you'll get a Makefile.in file containing several hundred lines of makefile code. But if you add additional makefile syntax to your Makefile.am files, this code will be transferred to the most functionally correct location in the resulting Makefile.in. In fact, you can (if you wish) write pure make syntax in your Makefile.am files, and they'll work just fine (as long as you actually write them correctly, that is). This pass-through feature gives you the power and flexibility to extend Automake's functionality with your project's own special requirements.

Aclocal

The aclocal utility is actually documented by the GNU manuals as a temporary work-around for a certain lack of flexibility in Autoconf. Autoconf was designed and written first, and then a few years later, the idea for Automake was conceived as an add-on for Autoconf. But Autoconf was really not designed to be extensible on the scale required by Automake.

Automake adds an extensive set of macros to those provided by Autoconf. The originally documented method for adding user-defined macros to an Autoconf project was to create a file called aclocal.m4 in the same directory as configure.ac. Any user-provided extension macros were to be placed in this file, and Autoconf would automatically read it while processing configure.ac. From the perspective of the Automake designers, this existing extension mechanism was too good to pass up. But requiring the user to add an `m4_include` line to aclocal.m4 seemed a bit brittle. Instead, the aclocal utility was designed to create a project's aclocal.m4 file, containing all the required Automake macros. Since Automake's aclocal utility basically took over aclocal.m4 for its own purposes, it was also designed to read a new user-provided macro file called acinclude.m4.
Essentially, aclocal's job is to create an aclocal.m4 file by consolidating various macro files from installed Autotool packages and user-specified locations, such that Autoconf can find them all in one place.

For the sake of modularity, the Autoconf manual is still unaware of the aclocal utility--for the most part. The current revision of the manual rants a bit on the subject of where aclocal functionality should actually be. Automake's manual originally suggested that you should rename aclocal.m4 to acinclude.m4 when adding Automake to an existing Autoconf project. This method is still followed rigorously in new projects.

However, the latest documentation from both sets of tools suggests that the entire aclocal/acinclude paradigm is now obsolete, in favor of a newer method of specifying a *directory* containing m4 macro files. The current recommendation is that you create a directory in your project directory called simply `m4` (`acinclude` seems more appropriate to this author), and add macros in the form of individual `.m4` files to this directory. All files in this directory will be gathered into aclocal.m4 before Autoconf processes your configure.ac file. Ultimately, aclocal will be replaced by functionality in Autoconf itself. (Given the fairly complex nature of aclocal functionality, and given that most of the other tools are already written in perl, I'm guessing that Autoconf will be rewritten in perl, at this point.)

Figure 2: Aclocal data flow diagram

With aclocal behind us, it should be more apparent now why the aclocal.m4 box in the Autoconf data flow diagram of Figure 1 above couldn't decide which color it should be. When used without Automake and Libtool, the aclocal.m4 file is written by hand, but when used in conjunction with Automake and Libtool, the file is generated by the aclocal utility, and acinclude.m4 is used to provide project-specific macros.

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 package tries to address.

There's a significant amount of visible functionality in Unix and Unix-like platforms that is the same from one platform to another. However, one very significant difference is how shared libraries are built, named and managed. Some platforms don't even provide native shared libraries (although it's rare these days). Some platforms name their libraries `libsomething.so`, while others use `something.o`. Some use `libsomething.a`, while others use `libsomething.sa`. Some platforms provide libdl (dlopen/dlsym/dlclose) to allow software to dynamically load and access library functionality at runtime. Others provide other mechanisms--or none at all.
All of these differences have been carefully considered by the authors of the Libtool project. Dozens of platforms are currently supported by Libtool, and adding support for new platforms is done via the open source way--someone who cares (and knows how) supplies a patch to the Libtool mailing list, and the maintainers look it over and apply it to the source code for the next release.

Libtool not only provides a set of Autoconf macros that hide library naming differences in makefiles, but it also provides an optional library of dynamic loader functionality that can be added to your programs, allowing you to write more portable *runtime* dynamic shared object management code.
The libool package provides the following programs, libraries and header files:

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

The libtool shell script is a generic version of Libtool designed to be used by programs on your platform. There's nothing specific to a project in this particular copy of libtool.

Libtoolize

The libtoolize shell script is used to prepare your project to use Libtool. In reality, libtoolize generates a custom version of the libtool script in your project directory. This script is then executed at the appropriate time by Automake-generated makefiles.

The Libtool C API--ltdl

The Libtool package also provides the ltdl library and header files, which provide a consistent run-time 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 from one platform to another.

The following data flow diagram illustrates the interaction between Automake and Libtool scripts and input files to create products used by users to configure and build your project:


Figure 3: Automake and Libtool data flow diagram

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

Building your package

While, as maintainer, you probably build your software packages a lot more often than do your users, you also have the advantage of being intimately familiar with your project's components, architecture and build system. That's why you ought to be concerned that your users' build experience is much simpler than yours. (And it wouldn't hurt a bit if you got some benefit from this concern, as well.)

Running configure

Once the Autotools have finished their work, you're left with a shell script called configure, and one or more Makefiles.in files. These product files are intended to be packages with project release distribution packages. Your users download these packages, unpack them, and run configure and make. The configure script generates Makefiles from the Makefile.in files. It also generates a config.h header file from the config.h.in file built by autoheader.

So why didn't the Autotools just generate the makefiles directly to be shipped with your release? One reason is that without makefiles, you can't run make. This means that you're forced to run configure first, after you download and unpack a project distribution package. Makefile.in files are nearly identical to the makefiles you might write by hand, except that you didn't have to. And they do a lot more than most people are willing to hand code into a set of makefiles. Another reason is that the configure script may then insert platform-characteristics and user-specified optional features directly into your makefiles, making them more specifically tailored to the platforms on which they are being used.

The following diagram illustrates the interaction between configure and the scripts that it executes during the build process to create your Makefiles and your config.h header file:


Figure 4: Configure script data flow diagram

The configure script appears to have this weird sort of incestuous relationship with another script called config.status. I'll bet you've always thought that your configure script generated your makefiles. As it turns out, the only file (besides a log file) that configure generates is config.status. The configure script's function is to determine platform characteristics and features available, as specified in configure.ac. Once it has this information, it generates config.status such that it contains all of the check results, and then calls it. The newly generated config.status file uses the check information (now embedded within it) to generate platform-specific config.h and makefiles, as well as any other files specified for instantiation in configure.ac. As the double ended red arrow shows, config.status can also call configure. When used with the --recheck option, config.status will call configure with the same command line options with which it was originally generated.

The configure script also generates a log file called config.log, which contains very useful information about why a particular execution of configure failed on your user's platform. As maintainer, you can use this information to help you debug user problems. Just ask them to send you their config.log file. The problem is often in plain sight. Another nice feature of config.log is that it logs how configure was executed--which command line options were used.

From a user perspective, this could be really handy, as he comes back from a long vacation, and can't remember what options he used to generate the project build directory. But Autoconf-generated configure scripts make it even simpler than this. If you need to re-generate makefiles and config.h header files for some reason, just type ./config.status in the project build directory. The output files will be generated using the same options originally used to generate the config.status file.

Remote build directories

A little-known feature of Autotools build environments is that they need not be generated within a project source directory tree. That is, a user may execute configure remotely, and generate a full build environment within a remote build directory.

In the following example, Joe User downloads doofabble 3.0 and unpacks it. Then he creates two sibling directories called doofabble-3.0.debug and doofabble-3.0.release. He cd's into doofabble-3.0.debug, executes doofabble-3.0's configure script remotely with a doofabble-specific debug option, and then runs make. Finally, he switches over to the doofabble-3.0.release directory and does the same thing, this time running configure without the debug option enabled:

```
$ tar -zxvf doofabble-3.0.tar.gz

$ mkdir doofabble-3.0.debug

$ cd doofabble-3.0.debug

$ ../doofabble-3.0/configure --enable-debug

$ make

...

$ cd ..

$ mkdir doofabble-3.0.release

$ cd doofabble-3.0.release

$ ../doofabble-3.0/configure
```

```
$ make

...
```

Users don't often care about remote build functionality because all they generally want to do is configure, make and install your code on their own platforms. Maintainers, on the other hand should find remote build functionality very useful, as it allows them to, 1) maintain a reasonably pristine source tree, and 2) maintain multiple build environments for their project, each with potentially complex configuration options. Rather than reconfigure a single build environment, they may simply switch between build directories configured in multiple different ways.

Running make

Finally, you run make. Just plain old make. In fact, the Autotools designers went to a LOT of trouble to ensure that you didn't need any special version or brand of make. You don't need GNU make--you can use Solaris make, or BSD Unix make if you wish (read, "if you must").

The following diagram depicts the interaction between the make utility and the generated makefiles during the build process to create your project products:

Figure 5: Make data flow diagram

This diagram shows make running several generated scripts, but these are all really ancillary to the make process.

Summary

In this chapter I've presented a high-level overview of the Autotools to give you a feel for how everything ties together.

In the next chapter, we'll begin creating a hand-coded build system for a toy project. The idea is that you'll become familiar with the requirements of a reasonable build system, and how much can be done for you by the Autotools.

Too many developers these days start out with the Autotools, not having aquired through the "school of hard knocks" the experience to know what it's really doing for them. This can lead to frustration, and a negative attitude. In the next chapter, you'll become familiar with the rationale for a lot of the original design of the Autotools. In understanding this background information, my hope is that any potential negative bias you may already have for the Autotools will be tempered a bit.

‹ Autotools: a practitioner's guide to Autoconf, Automake and Libtool up Chapter 2: Project management and the GNU coding standards ›

Chapter 2: Project management and the GNU coding standards

Fri, 2008-05-16 11:16 -- John Calcote

In Chapter 1, I gave a brief overview of the Autotools and some of the resources that are currently available to help reduce the learning curve. In this chapter, we're going to step back a little and examine project organization techniques that are applicable to all projects, not just those whose build system is managed by the Autotools.

This chapter has downloads!
When you're done reading this chapter, you should be familiar with the common `make` targets, and why they exists. You should also have a solid understanding of why projects are organized the way they are. Trust me--by the time you finish this chapter, you'll already be well on your way to a solid understanding of the GNU Autotools.
The information provided by this chapter comes primarily from two sources:

- The GNU Coding Standards Document
- The Filesystem Hierarchy Standard

In addition, you may find the GNU make manual very useful, if you'd like to brush up on your `make` syntax:

- The GNU Make Utility Manual

Creating a new project directory structure

There are two questions to ask yourself when setting up a new open source software (OSS) project build system:

- What platforms will I target?
- What do my users expect?

The first is an easy question to answer - you get to decide, but don't be too restrictive. Free software projects become great due to the number of people who've adopted them. Limiting the number of platforms arbitrarily is the direct equivalent of limiting the number of users. Now, why would you want to do that?!

The second question is more difficult, but not unsolvable. First, let's narrow the scope to something managable. We really mean to say, "What do my users expect of my build system?" A common approach for many OSS developers of determining these expectations is to download, unpack, build and install about a thousand different packages. You think I'm kidding? If you do this, eventually, you will come to know intuitively what your users expect of your build system. Unforutunately, package configuration, build and install processes vary so far from the "norm" that it's difficult to come to a solid conclusion about what the norm really is when using this technique.

A better way is to go directly to the source of the information. Like many developers new to the OSS world, I didn't even know there *was* a source of such information when I first started working on OSS projects. As it turns out, the source is quite obvious, after a little thought: The Free Software Foundation (FSF), better known as the GNU project. The FSF has published a document called The GNU Coding Standards, which covers a wide variety of topics related to writing, publishing and distributing free software--specifically for the FSF. Most non-GNU free software projects align themselves to one degree or another with the GNU Coding Standards. Why? Well...just because they were there first. And because their ideas make sense, for the most part.

Project structure

We'll start with a simple example project, and build on it as we continue our exploration of source-level software distribution. OSS projects generally have some sort of catchy name--often they're named after some past hero or ancient god, or even some made-up word--perhaps an acronym that can be pronounced like a real word. I'll call this the jupiter project, mainly because that way I don't have to come up with functionality that matches my project name! For jupiter, I'll create a project directory structure something like this:

```
$ cd projects

$ mkdir -p jupiter/src

$ touch jupiter/Makefile

$ touch jupiter/src/Makefile

$ touch jupiter/src/main.c

$ cd jupiter

$
```

Woot! One directory called `src`, one C source file called `main.c`, and a makefile for each of the two directories. Minimal yes, but hey, this is a new project, and everyone knows that the key to a successful OSS project is evolution, right? Start small and grow as needed (and, as you have time and inclination).

We'll start with support for the most basic of targets in any software project: `all` and `clean`. As we progress, it'll become clear that we need to add a few more important targets to this list, but for now, these will get us going. The top-level Makefile does very little at this point, merely passing requests for `all` and `clean` down to `src/Makefile` recursively. In fact, this is a fairly common type of build system, known as a *recursive build system*. Here are the contents of each of the three files in our project:

`Makefile`

```
all clean jupiter:

	$(MAKE) -C src $@
```

`src/Makefile`

```
all: jupiter



jupiter: main.c

	gcc -g -O0 -o $@ $+
```

```
clean:
	-rm jupiter
```

src/main.c

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


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

At this point, you may need to stop and take a refresher course in `make` syntax. If you're already pretty well versed on `make`, then you can skip the sidebar entitled, "Some makefile basics". Otherwise, give it a quick read, and then we'll continue building on this project.

Some makefile basics

For those like myself who use `make` *only when they have to, it's often difficult to remember exactly what goes where in a makefile. Well, here are a few things to keep in mind. Besides comments, which begin with a HASH mark, there are only three types of entities in a makefile:*

- variable assignments
- rules
- commands

NOTE: There are a half-dozen other types of constructs in a makefile, including conditional statements, directives, extension rules, pattern rules, function variables, include statements, etc. For the purposes of this chapter, we need not go into these constructs. This doesn't mean these other constructs are unimportant. On the contrary, they are very useful if you're going to write your own complex build system by hand. Our purpose here is to gain the background necessary for an understanding of the GNU Autotools, so I'll cover only that portion of `make` *necessary to accomplish this goal. If you wish to have a much broader education on* `make` *syntax, please refer to the GNU make manual. Furthermore, if you wish to become a* `make` *expert, be prepared to spend a good deal of time on the project--there's much more to the* `make` *utility than is initially apparent on the surface.*

Commands always start with a TAB character. Any line in a makefile beginning with a TAB character is ALWAYS considered by `make` *to be a command. A list of one or more commands should always be associated with a preceeding rule.*

NOTE: The fact that commands are required to be prefixed with an essentially invisible character is one of the most frustrating aspects of makefile syntax to both neophites and experts alike. The error messages generated by the legacy Unix `make` utility when a required TAB is missing, or when an unintentional TAB is inserted are obscure at best. As mentioned earlier, GNU make does a better job with such error messages these days. Nonetheless, be careful to use TAB characters properly in your makefiles--only before commands, which in turn immediately follow rules.
The general layout of a makefile is:

```
var1=val1

var2=val2

...

rule1

        cmd1a

        cmd1b

        ...

rule2

        cmd2a

        cmd2b

        ...
```

Variable assignments may take place at any point in the makefile, however you should be aware that `make` reads each makefile twice. The first pass gathers variables and rules into tables, and the second pass resolves dependencies defined by the rules. So regardless of where you put your variable definitions, `make` will act as though they'd all been declared at the top, in the order you specified them throughout the makefile.
Furthermore, `make` binds variable references to values at the very last minute--just before referencing commands are passed to the shell for execution. So, in general, variables may be assigned values by reference to other variables that haven't even been assigned yet. Thus, the order of variable assignment isn't really that important.
The `make` utility is a rule-based command engine. The rules indicate when and which commands should be executed. When you prefix a line with a TAB character, you're telling `make` that you want it to execute these statements from a shell according to the rules specified on the line above.
Of the remaining lines, those containing an EQUAL sign are variable definitions. Variables in makefiles are nearly identical to shell or environment variables. In Bourne shell syntax, you'd reference a variable in this manner: `${my_var}`. In a makefile, the same syntax applies, except you would use parentheses instead of french braces: `$(my_var)`. As in shell syntax, the delimiters are optional, but

should be used to avoid ambiguous syntax when necessary. Thus, $my_var is functionally equivalent to $(my_var).

One caveat: If you ever want to use a shell variable inside a make command, you need to escape the DOLLAR sign by doubling it. For instance, $${shell_var}. This need arises occasionally, and it nearly always catches me off-guard the first time I use it in a new project.

Variables may be defined and used anywhere within a makefile. By default, make will read the entire process environment into the make variable table before processing the makefile. Thus, you can access any environment variables as if they were defined in the makefile itself. Note however, that variables set in the makefile will override those obtained from the environment. In general, it's a good idea not to depend on environment variables in your build process, although it's okay to use certain variables conditionally, if they're present. In addition, make defines several useful variables of its own, such as the MAKE variable, whose value is the file system path used to invoke the current make process.

Lines in my example makefiles that are not variable assignments (don't contain an EQUAL sign), and are not commands (are not prefixed with a TAB character), are all rules of one type or another. The rules used in my examples are known as "common" make rules, containing a single COLON character. The COLON character separates targets on the left from dependencies on the right. Targets are products--generally file system entities that can be produced by running one or more commands, such as a C compiler or a linker. Dependencies are source objects, or objects from which targets may be created. These may be computer language source files, or anything really that can be used by a command to generate a target object.

For example, a C compiler takes dependency main.c as input, and generates target main.o. A linker takes dependency main.o as input, and generates a named executable target, jupiter, in these examples:


Figure 6: Compile and link process

The make utility implements some fairly complex logic to determine when a rule should be run based on whether the target exists or is older than its dependencies, but the syntax is trivial enough:

```
jupiter: main.o

        ld main.o ... -o jupiter



main.o: main.c

        gcc -c -g -O2 -o main.o main.c
```

This sample makefile contains two rules. The first says that jupiter depends on main.o, and the second says that main.o depends on main.c. Ultimately, of course, jupiter depends on main.c, but main.o is a necessary intermediate dependency in this case, because there are two steps to the process--compile and link--with an intermediate result in between. For each rule, there is an associated list of commands that make uses to build the target from the list of dependencies.

Of course, there is an easier way in the case of this example--gcc (as with most compilers) will call the linker for you--which, as you can probably tell from the elipsis in my example above, is very desirable. This alleviates the need for one of the rules, and provides a convenient way of adding more dependent files to the single remaining rule:

```
sources = main.c print.c display.c



jupiter: $(sources)

        gcc -g -O2 -o jupiter $(sources)
```

NOTE: I should point out that using a single rule and command to process both steps is possible in this case because of the triviality of the example. In larger projects, skipping from source to executable in a single step is not possible. In these cases, using the compiler to call the linker can ease the burden in the second stage of determining all of the system objects that need to be linked into an application. And, in fact, this very technique is used quite often on Unix-like systems.

In this example, I've added a `make` variable to reduce redundancy. We now have a list of source files that is referenced in two places. But, it seems a shame to be required to reference this list twice in this manner, when the `make` utility knows which rule and which command it's dealing with at any moment during the process. Additionally, there may be other objects in the dependency list that are not in the `sources` variable. It would be nice to be able to reference the entire dependency list without duplicating that list.

As it happens, there are various "automatic" variables that can be used to reference portions of the controlling rule during the execution of a command. For example `$(@)` (or the more common syntax `$@`) references the current target, while `$+` references the current list of dependencies:

```
sources = main.c print.c display.c



jupiter: $(sources)

        gcc -g -O2 -o $@ $+
```

If you enter "`make`" on the command line, the `make` utility will look for the first target in a file named "`Makefile`" in the current directory, and try to build it using the rules defined in that file. If you specify a different target on the command line, `make` will attempt to build that target instead.

Targets need not be files only. They can also be so-called "phony targets", defined for convenience, as in the case of `all` and `clean`. These targets don't refer to true products in the file system, but rather to particular outcomes--the directory is "cleaned", or "all" desirable targets are built, etc.

In the same way that dependencies may be listed on the right side of the COLON, rules for multiple targets with the same dependencies may be combined by listing targets on the left side of the COLON, in this manner:

```
all clean jupiter:

        $(MAKE) -C src $@
```

The -C command-line option tells `make` to change to the specified directory before looking for a makefile to run.
GNU Make is significantly more powerful than the original Unix `make` utility, although completely backward compatible, as long as GNU extensions are avoided. The GNU Make manual is available online. O'Reilly has an excellent book on the original Unix `make` utility and all of its many nuances. They also have a more recent book written specifically for GNU make that covers GNU Make extensions.

Creating a source distribution archive

It's great to be able to type "`make all`" or "`make clean`" from the command line to build and clean up this project. But in order to get the jupiter project source code to our users, we're going to have to create and distribute a source archive.
What better place to do this than from our build system. We could create a separate script to perform this task, and many people have done this in the past, but since we have the ability, through phony targets, to create arbitrary sets of functionality in `make`, and since we already have this general purpose build system anyway, we'll just let `make` do the work for us.
Building a source distribution archive is usually relegated to the `dist` target, so we'll add one. Normally, the rule of thumb is to take advantage of the recursive nature of the build system, by allowing each directory to manage its own portions of a global process. An example of this is how we passed control of building jupiter down to the `src` directory, where the jupiter source code is located. However, the process of building a compressed archive from a directory structure isn't really a recusive process--well, okay, yes it is, but the recursive portions of the process are tucked away inside the `tar` utility. This being the case, we'll just add the `dist` target to our top-level makefile:
`Makefile`

```
package = jupiter

version = 1.0

tarname = $(package)

distdir = $(tarname)-$(version)



all clean jupiter:
```

```
        $(MAKE) -C src $@

dist: $(distdir).tar.gz

$(distdir).tar.gz: $(distdir)
        tar chof - $(distdir) |\
          gzip -9 -c >$(distdir).tar.gz
        rm -rf $(distdir)

$(distdir):
        mkdir -p $(distdir)/src
        cp Makefile $(distdir)
        cp src/Makefile $(distdir)/src
        cp src/main.c $(distdir)/src

.PHONY: all clean dist
```

In this version of the top-level `Makefile`, we've added a new construct, the `.PHONY` rule. At least it seems like a rule--it contains a COLON character, anyway. The `.PHONY` rule is a special kind of rule called a "dot-rule", which is built into `make`. The `make` utility understands several different dot-rules. The purpose of the `.PHONY` rule is simply to tell `make` that certain targets don't generate file system objects, so `make` won't go looking for product files in the file system that are named after these targets. Normally, the `make` utility determines which commands to run by comparing the time stamps of the associated rule products to those of their dependencies in the file system, but phony targets don't have associated file system objects.

We've added the new `dist` target in the form of three rules for the sake of readability, modularity and maintenance. This is a great rule of thumb to following in any software engineering process: Build large processes from smaller ones, and reuse the smaller processes where it makes sense to do so.

The `dist` target depends on the existance of the ultimate goal, a source-level compressed archive package, `jupiter-1.0.tar.gz`--also known as a "tarball". I've added a `make` variable for the version number to ease the process of updating the project version later, and I've used another variable for the package name for the sake of possibly porting this makefile to another project. I've also logically split the functions of package name and tar name, in case we want them to be different later--the default tar name is the package name. Finally, I've combined references to these variables into a `distdir` variable to reduce duplication and complexity in the makefile.

The rule that builds the tarball indicates how this should be done with a command that uses the `gzip` and `tar` utilities to create the file. But, notice also that the rule has a dependency--the directory to be archived. We don't want everything in our project directory hierarchy to go into our tarball--only exactly those files that are necessary for the distribution. Basically, this means any file required to build and install our project. We certainly don't want object files and executables from our last build attempt to end up in the archive, so we have to build a directory containing exactly what we want to ship. This pretty much mandates the use of individual `cp` commands, unfortunately.

Since there's a rule in the makefile that tells how this directory should be created, `make` runs the commands for this rule *before* running the commands for the current rule. The `make` utility runs rules to build dependencies recursively until the requested target's commands can be run.

Forcing a rule to run

There's a subtle flaw in the `$(distdir)` target that may not be obvious, but it will rear its ugly head at the worst times. If the archive directory already exists when you type `make dist`, then `make` won't try to create it. Try this:

```
$ mkdir jupiter-1.0

$ make dist

tar chof - jupiter-1.0 | gzip -9 -c >jupiter-1.0...

rm -rf jupiter-1.0 &> /dev/null

$
```

Notice that the `dist` target didn't copy any files--it just built an archive out of the existing `jupiter-1.0` directory, which was empty. Our end-users would have gotten a real surpise when they unpacked this tarball!

The problem is that the `$(distdir)` target is a real target with no dependencies, which means that `make` will consider it up-to-date as long as it exists in the file system. We could add `$(distdir)` to the `.PHONY` rule, but this would be a lie--it's not a phony target, it's just that we want to force it to be rebuilt every time.

The proper way to ensure it gets rebuilt is to have it not exist before `make` attempts to build it. A common method for accomplishing this task to to create a true phony target that will run every time, and add it to the dependency chain at or above the `$(distdir)` target. For obvious reasons, a commonly used name for this sort of target is "`FORCE`":

`Makefile`

```
...

$(distdir).tar.gz: FORCE $(distdir)

        tar chof - $(distdir) |\

          gzip -9 -c >$(distdir).tar.gz

        rm -rf $(distdir)
```

```
$(distdir):
        mkdir -p $(distdir)/src
        cp Makefile $(distdir)
        cp src/Makefile $(distdir)/src
        cp src/main.c $(distdir)/src

FORCE:
        -rm $(distdir).tar.gz &> /dev/null
        -rm -rf $(distdir) &> /dev/null

.PHONY: FORCE all clean dist
```

The `FORCE` rule's commands are executed every time because `FORCE` is a phony target. By making `FORCE` a dependency of the tarball, we're given the opportunity to delete any previously created files and directories before `make` begins to evaluate whether or not these targets' commands should be executed. This is really much cleaner, because we can now remove the "pre-cleanup" commands from all of the rules, except for `FORCE`, where they really belong.

There are actually more accurate ways of doing this--we could make the `$(distdir)` target dependent on all of the files in the archive directory. If any of these files are newer than the directory, the target would be executed. This scheme would require an elaborate shell script containing `sed` commands or non-portable GNU make functions to replace file paths in the dependency list for the copy commands. For our purposes, this implementation is adequate. Perhaps it would be worth the effort if our project were huge, and creating an archive directory required copying and/or generating thousands of files.

The use of a leading DASH character on some of the `rm` commands is interesting. A leading DASH character tells `make` to not care about the status code of the associated command. Normally `make` will stop execution with an error message on the first command that returns a non-zero status code to the shell. I use a leading DASH character on the `rm` commands in the `FORCE` rule because I want to delete previously created product files that may or may not exist, and `rm` will return an error if I attempt to delete a non-existent file. Note that I explicitly did NOT use a leading DASH on the `rm` command in the `$(distdir)` rule. This is because this `rm` command must succeed, or something is very wrong, as the preceeding command should have created a tarball from this directory.

Another such leading character that you may encounter is the ATSIGN (`@`) character. A command prefixed with an ATSIGN tells `make` not to print the command as it executes it. Normally `make` will print each command as it's executed. A leading ATSIGN tells `make` that you don't want to see this command. This is a common thing to do on `echo` statements--you don't want `make` to print `echo` statements, because then your message will be printed twice, and that's just ugly.

Automatically testing a distribution

The rule for building the archive directory is the most frustrating of any in this makefile--it contains commands to copy files *individually* into the distribution directory. What a sad shame! Everytime we change the file structure in our project, we have to update this rule in our top-level makefile, or we'll break our `dist` target.

But, there's nothing to be done for it. We've made the rule as simple as possible. Now, we just have to remember to manage this process properly. But unfortunately, breaking the `dist` target is not the worst thing that could happen if we forget to update the `distdir` rule's commands. The `dist` target may continue to *appear* to work, but not actually copy all of the required files into the tarball. This will cause us some embarassment when our users begin to send us emails asking why our tarball doesn't build on their systems.

In fact, this is a far more common possibility than that of breaking the `dist` target, because the more common activity while working on a project is to *add* files to the project, not move them around or delete them. New files will not be copied, but the `dist` rule won't notice the difference.

If only there were some way of unit-testing this process. As it turns out, there is a way of performing a sort of self-check on the `dist` target. We can create yet another phony target called "`distcheck`" that does exactly what our users will do--unpack the tarball, and build the project. We can do this in a new temporary directory. If the build process fails, then the `distcheck` target will break, telling us that we forgot something crucial in our distribution.

`Makefile`

```
...

distcheck: $(distdir).tar.gz

        gzip -cd $+ | tar xvf -

        $(MAKE) -C $(distdir) all clean

        rm -rf $(distdir)

        @echo "*** Package $(distdir).tar.gz\

          ready for distribution."

...

.PHONY: FORCE all clean dist distcheck
```

Here, we've added the `distcheck` target to the top-level makefile. Since the `distcheck` target depends on the tarball itself, it will first build a tarball using the same targets used by the `dist` target. It will then execute the `distcheck` commands, which are to unpack the tarball it just built and run "`make all clean`" on the resulting directory. This will build both the `all` and `clean` targets, successively. If that process succeeds, it will print out a message, telling us that we can sleep well knowing that our users will probably not have a problem with this tarball.

Now all we have to do is remember to run "`make distcheck`" *before* we post our tarballs for public distribution!

Unit testing anyone?

Some people think unit testing is evil, but really--the only honest rationale they can come up with for not doing it is laziness. Let's face it--proper unit testing is hard work, but it pays off in the end. Those who do it have learned a lesson (usually as children) about the value of delayed gratification.

A good build system is no exception. It should encorporate proper unit testing. The commonly used target for testing a build is the `check` target, so we'll go ahead and add the `check` target in the usual manner. The test should probably go in `src/Makefile` because jupiter is built in `src/Makefile`, so we'll have to pass the `check` target down from the top-level makefile.

But what commands do we put in the `check` rule? Well, jupiter is a pretty simple program--it prints out a message, "Hello from <path>jupiter!", where <path> is variable, depending on the location from which jupiter was executed. We could check to see that jupiter actually does output such a string. We'll use the `grep` utility to test our assertion:

`Makefile`

```
...
all clean check jupiter:
        $(MAKE) -C src $@
...
.PHONY: FORCE all clean check dist distcheck
```

`src/Makefile`

```
...
check: all
        ./jupiter | grep "Hello from .*jupiter!"
        @echo "*** ALL TESTS PASSED ***"
...
.PHONY: all clean check
```

Note that `check` is dependent on `all`. We can't really test our products unless they've been built. We can ensure they're up to date by creating such a dependency. Now `make` will run commands for `all` if it needs to before running the commands for `check`.

There's one more thing we could do to enhance our build system a bit. We can add the `check` target to the `make` command in our `distcheck` target. Adding it right between the `all` and `clean` targets seems appropriate:

`Makefile`

```
...

distcheck: $(distdir).tar.gz

        gzip -cd $+ | tar xvf -

        $(MAKE) -C $(distdir) all check clean

        rm -rf $(distdir)

        @echo "*** Package $(distdir).tar.gz\

          ready for distribution."

...
```

Now, when we run "`make distcheck`", our entire build system will be tested before packaging is considered successful. What more could you ask for?!

Installing products

Well, we've now reached the point where our users' experiences with our project should be fairly painless--even pleasant, as far as building the project is concerned. Our users will simply unpack the distribution tarball, change into the distribution directory, and type "`make`". It can't really get any simpler than that.

But still we lack one important feature--installation. In the case of the jupiter project, this is fairly trivial - there's only one executable, and most users could probably guess that this file should be copied into either the `/usr/bin` or `/usr/local/bin` directory. More complex projects, however could cause our users some real consternation when it comes to where to put user and system binaries, libraries, header files, and documentation, including man pages, info pages, pdf files, and README, INSTALL and COPYRIGHT files. Do we really want our users to have to figure all that out?

I don't think so. So we'll just create an `install` target that manages putting things where they go, once they're built properly. Why not just make installation part of the `all` target? A few reasons, really. First, build and installation are separate logical concepts. Remember the rule: Break up large processes into smaller ones and reuse the smaller ones where you can. The second reason is a matter of rights. Users have rights to build in their own home directories, but installation often requires root-level rights to copy files into system directories. Finally, there are several reasons why a user may wish to build, but not install.

While creating a distribution package may not be an inherently recursive process, installation certainly is, so we'll allow each subdirectory in our project to manage installation of its own components. To do this, we need to modify both makefiles. The top-level makefile is easy. Since there are no products to be installed in the top-level directory, we'll just pass on the responsibility to `src/Makefile` in the usual way:

`Makefile`

```
...

all clean check install jupiter:
```

```
        $(MAKE) -C src $@

...

.PHONY: FORCE all clean check dist distcheck

.PHONY: install
```

src/Makefile

```
...

install:

        cp jupiter /usr/bin

        chown root:root /usr/bin/jupiter

        chmod +x /usr/bin/jupiter



.PHONY: all clean check install
```

In the top-level makefile, we've added `install` to the list of targets passed down to `src/Makefile`. In both files we've added `install` to the phony target list.

As it turns out, installation was a bit more complex than simply copying files. If a file is placed in the `/usr/bin` directory, then the root user should own it so that only the root user can delete or modify it. Additionally, we should ensure that the jupiter binary is executable, so we use the `chmod` command to set the mode of the file to executable. This is probably redundant, as the linker ensures that jupiter gets created as an executable file, but it never hurts to be safe.

Now our users can just type the following sequence of commands, and have our project built and installed with the correct system attributes and ownership on their platforms:

```
$ tar -zxvf jupiter-1.0.tar.gz

$ cd jupiter-1.0

$ make all

$ sudo make install
```

All of this is well and good, but it could be a bit more flexible with regard to *where* things get installed. Some of our users may be okay with having jupiter installed into the `/usr/bin` directory. Others are going to ask us why we didn't put it into the `/usr/local/bin` directory--after all, this is a common convention. Well, we could change the target directory to `/usr/local/bin`, but then others will ask us why we didn't just put it into the `/usr/bin` directory. This is the perfect situation for a little command-line flexibility.

Another problem we have with these makefiles is the amount of stuff we have to do to install files. Most Unix systems provide a system-level program called "`install`", which allows a user to specify, in an intelligent manner, various attributes of the files being installed. The proper use of this utility could simplify things a bit. While we're adding location flexibility, I'll just go ahead and add the use of the `install` utility, as well:

`Makefile`

```
...

export prefix=/usr/local



all clean install jupiter:

        $(MAKE) -C src $@

...
```

`src/Makefile`

```
...

install:

        install -d $(prefix)/bin

        install -m 0755 jupiter $(prefix)/bin

...
```

If you're astute, you may have noticed that I've declared and assigned the `prefix` variable in the top-level makefile, but I've *referenced* it in `src/Makefile`. This is possible because I used the `export` modifier in the top-level makefile to export this `make` variable to the shell that `make` spawns when it executes itself in the `src` directory. This is a nice feature of `make` because it allows us to define all of our user variables in one obvious location--at the top of the top-level makefile.

I've now declared the `prefix` variable to be `/usr/local`, which is very nice for those who want jupiter to be installed in `/usr/local/bin`, but not so nice for those who just want it installed in `/usr/bin`. Fortunately, `make` allows the definition of `make` variables on the command line, in this manner:

```
$ sudo make prefix=/usr install

...
```

Variables defined on the command line *override* those defined in the makefile. Thus, users who want to install jupiter into their `/usr/bin` directory now have the option of specifying this on the `make` command line when they install `jupiter`.

Actually, with this system in place, our users may install jupiter into *any* directory they choose, including a location in their home directory, for which they do not need additional rights granted. This is, in fact, the reason for the addition of the `mkdir -p` command. We don't actually know *where* the user is going to install jupiter now, so we have to be prepared for the possiblity that the location may not yet exist.

A bit of trivia about the `install` utility--it has the interesting property of changing the ownership of any file it copies to the owner and group of the containing directory. So it automatically sets the owner and group of our installed files to `root:root` if the user tries to use the default `/usr/local` prefix, or to the user's id and group if she tries to install into a location within her home directory. Nice, huh?

Uninstalling a package

What if a user doesn't like our package after it's been installed, and she just wants to get it off her system? This is fairly likely with the jupiter package, as it's rather useless and takes up valuable space in her `bin` directory. In the case of your projects however, it's more likely that she wants to install a newer version of your project cleanly, or she wants to change from the test build she downloaded from your website to a professionally packaged version of your project provided by her Linux distribution. We really should have an `uninstall` target, for these and other reasons:

`Makefile`

```
...

all clean install uninstall jupiter:

        $(MAKE) -C src $@

...

.PHONY: FORCE all clean dist distcheck

.PHONY: install uninstall
```

`src/Makefile`

```
...

uninstall:

        -rm $(prefix)/bin/jupiter



.PHONY: all clean check install uninstall
```

And, again, this particular target will require root-level rights if the user is using a system prefix, such as `/usr` or `/usr/local`. The list of things to maintain is getting a out of hand, if you ask me. We now have two places to update when changing our installation processes--the `install` and `uninstall` targets. Unfortunately, this is really about the best we can hope for when writing our own makefiles, without resorting to fairly complex shell script commands. Hang in there--in Chapter 6, I'll show you how this example can be rewritten in a much simpler way using Automake.

Finally, while we're at it, let's add testing the `install` and `uninstall` targets to our `distcheck` target:

`Makefile`

```
...

distcheck: $(distdir).tar.gz

        gzip -cd $+ | tar xvf -

        $(MAKE) -C $(distdir) all check

        $(MAKE) -C $(distdir) prefix=\

         $${PWD}/$(distdir)/_inst install uninstall

        $(MAKE) -C $(distdir) clean

        rm -rf $(distdir)

        @echo "*** Package $(distdir).tar.gz\

          ready for distribution."

...
```

To do this properly, I had to break up the `$(MAKE)` commands into three different steps, so that we could add the proper prefix to the `install` and `uninstall` targets without affecting the other targets. I'll have more to say on this topic in a few minutes.

Note also that I used a double DOLLAR sign on the `$${PWD}` variable reference. This was done in order to ensure that `make` passed the reference to the shell with the rest of the command line. I wanted this variable to be dereferenced by the shell, rather than the `make` utility. Technically, I didn't have to do this because the `PWD` variable was initialized for `make` from the environment, but it serves as a good example of this process.

The Filesystem Hierarchy Standard

By the way, where am I getting these directory names from? What if some Unix system out there doesn't use `/usr` or `/usr/local`? Well, in the first place, this is another reason for providing the `prefix` variable--to handle those sorts of situations. However, most Unix and Unix-like systems nowadays follow the Filesystem Hierarchy Standard (FHS), as closely as possible. The FHS defines a number of "standard places", including the following root-level directories:

- `/bin`
- `/etc`
- `/home`
- `/opt`

- `/sbin`
- `/srv`
- `/tmp`
- `/usr`
- `/var`

This list is not exhaustive. I've only mentioned the ones most relevant to our purposes. In addition, the FHS defines several standard locations beneath these root-level directories. For instance, the `/usr` directory should contain the following sub-directories:

- `/usr/bin`
- `/usr/include`
- `/usr/lib`
- `/usr/local`
- `/usr/sbin`
- `/usr/share`
- `/usr/src`

The `/usr/local` directory should contain a structure very similar to the `/usr` directory structure, so that if the `/usr/bin` directory (for instance) is an NFS mount, then `/usr/local/bin` (which should always be local) may contain local copies of some programs. This way, if the network is down, the system may still be usable, to some degree.

Not only does the FHS define these standard locations, but it also explains in fair detail what they are for, and what types of files should be kept there. All in all, the FHS leaves just enough flexibility and choice to you as a project maintainer to keep your life interesting, but not enough to make you lose sleep at night, wondering if you're installing your files in the right places.

Before I found out about the FHS, I relied on my personal experience to decide where files should be installed in my projects. Mostly I was right, because I'm a careful guy, but I have gone back to some of my past projects with a bit of chagrin and changed things, once I read the FHS document. I heartily recommend you become thoroughly familiar with this document if you seriously intend to develop Unix software.

Supporting standard targets and variables

In addition to those I've already mentioned, the GNU Coding Standards document lists some important targets and variables that you should support in your projects, mainly because everyone else does and your users will expect them.

Some of the chapters in the GNU Coding Standards should be taken with a grain of salt (unless you're actually working on a GNU sponsored project, in which case you're probably not reading this book because you need to). For example, you probably won't care much about the C source code formatting suggestions in Chapter 5. Your users certainly won't care, so you can use whatever source code formatting style you wish.

That's not to say that all of Chapter 5 is worthless. Sections 5.5 and 5.6, for instance, provide excellent information on C source code portability between POSIX-oriented platforms and CPU types. Section 5.8 gives some tips on using GNU software to internationalize your program. This is excellent material.

While Chapter 6 discusses documentation the GNU way, some sections of Chapter 6 describe various top-level text files found commonly in projects, such as the AUTHORS, NEWS, INSTALL, README and ChangeLog files. These are all bits that the well-read OSS user expects to see in any decent OSS project.

But, the *really* useful information in the GNU Coding Standards document begins in Chapter 7, "The Release Process". The reason why this chapter is so critical to you as an OSS project maintainer, is that it pretty much defines what your users will expect of your project's build system. Chapter 7 *is* the defacto-standard for user options provided by packages using source-level distribution.
Section 7.1 defines the configuration process, about which we haven't spent much time so far in this chapter, but we'll get to it. Section 7.2 covers makefile conventions, including all of the "standard targets" and "standard variables" that users have come to expect in OSS packages. Standard targets defined by the GNU Coding Standards document include:

- `all`
- `install`
- `install-html`
- `install-dvi`
- `install-pdf`
- `install-ps`
- `uninstall`
- `install-strip`
- `clean`
- `distclean`
- `mostlyclean`
- `maintainer-clean`
- `TAGS`
- `info`
- `dvi`
- `html`
- `pdf`
- `ps`
- `dist`
- `check`
- `installcheck`
- `installdirs`

Note that you don't need to support *all* of these targets, but you should consider supporting those which make sense for your project. For example, if you build and install HTML pages in your project, then you should probably consider supporting the `html` and `install-html` targets. Autotools projects support these, and more. Some of these are useful to users, while others are only useful to maintainers.
Variables that your project should support (as you see fit) include the following. I've added the default values for these variables on the right. You'll note that most of these variables are defined in terms of a few of them, and ultimately only one of them, `prefix`. The reason for this is (again) flexibility to the end user. I call these "prefix variables", for lack of a more standard name:

```
prefix          = /usr/local

exec-prefix     = $(prefix)

bindir          = $(exec_prefix)/bin

sbindir         = $(exec_prefix)/sbin

libexecdir      = $(exec_prefix)/libexec
```

```
datarootdir     = $(prefix)/share
datadir         = $(datarootdir)
sysconfdir      = $(prefix)/etc
sharedstatedir  = $(prefix)/com
localstatedir   = $(prefix)/var
includedir      = $(prefix)/include
oldincludedir   = /usr/include
docdir          = $(datarootdir)/doc/$(package)
infodir         = $(datarootdir)/info
htmldir         = $(docdir)
dvidir          = $(docdir)
pdfdir          = $(docdir)
psdir           = $(docdir)
libdir          = $(exec_prefix)/lib
lispdir         = $(datarootdir)/emacs/site-lisp
localedir       = $(datarootdir)/locale
mandir          = $(datarootdir)/man
manNdir         = $(mandir)/manN  (N = 1..9)
manext          = .1
manNext         = .N              (N = 1..9)
srcdir          = (compiled project root)
```

Autotools projects support these and other useful variables automatically. Projects that use Automake get these variables for free. Autoconf provides a mid-level form of support for these variables. If you write your own makefiles and build system, you should support as many of these as you use in your build and install processes.

To support the variables and targets that we've used so far in the jupiter project, we need to add the `bindir` variable, in this manner:

Makefile

```
...
export prefix = /usr/local
export exec_prefix = $(prefix)
export bindir = $(exec_prefix)/bin
...
```

src/Makefile

```
...
install:
	install -d $(bindir)
	install -m 0755 jupiter $(bindir)

uninstall:
	-rm $(bindir)/jupiter
...
```

Note that we have to export `prefix`, `exec_prefix` and `bindir`, even though we only use `bindir` explicitly in `src/Makefile`. The reason for this is that `bindir` is defined in terms of `exec_prefix`, which is itself defined in terms of `prefix`. So when `make` runs the install command, it will first resolve `bindir` to `$(exec_prefix)/bin`, and then to `$(prefix)/bin`, and finally to `/usr/local/bin`--`src/Makefile` obviously needs access to all three variables during this process.

How do such recursive variable definitions make life better for the end-user? The user can change the root install location from `/usr/local` to `/usr` by simply typing:

```
$ make prefix=/usr install
...
```

The ability to change these variables like this is particularly useful to a Linux distribution packager, who needs to install packages into very specific system locations:

```
$ make prefix=/usr sysconfdir=/etc install
...
```

Getting your project into a Linux distro

The dream of every OSS maintainer is that his or her project will be picked up by a Linux distribution. When a Linux "distro" picks up your package for distribution on their CD's and DVD's, your project will be moved magically from the realm of tens of users to that of tens of thousands of users--almost overnight.

By following the GNU Coding Standards with your build system, you remove many barriers to including your project in a Linux distro, because distro packagers (employees of the company, whose job it is to professionally package your project as RPM or APT packages) will immediately know what to do with your tarball, if it follows all the usual conventions. And, in general, packagers get to decide, based on needed functionality, and their feelings about your package, whether or not it should be included in their flavor of Linux.

Section 7.2.4 of the GNU Coding Standards talks about the concept of supporting "staged installations". This is a concept easily supported by a build system, but which if neglected, will almost always cause problems for Linux distro packagers.

Packaging systems such as the Redhat Package Manager (RPM) system accept one or more tarballs, a set of patches and a specification file (in the case of RPM, called an "rpm spec file"). The spec file describes the process of building and installing your package. In addition, it defines all of the products installed into the targeted installation directory hierarchy. The package manager software uses this information to install your package into a temporary directory, from which it pulls the specified binaries, storing them in a special binary archive that the package installation software (eg., `rpm`) understands.
To support staged installation, all you really need to do is provide a variable named "DESTDIR" in your build system that is a sort of super-prefix to all of your installed products. To show you how this is done, I'll add staged installation support to the jupiter project. This is so trivial, it only requires three changes to `src/Makefile`:

`src/Makefile`

```
...
install:
        install -d $(DESTDIR)$(bindir)
        install -m 0755 jupiter $(DESTDIR)$(bindir)

uninstall:
        -rm $(DESTDIR)$(bindir)/jupiter
...
```

As you can see, I've added the `$(DESTDIR)` prefix to the `$(bindir)` references in our install and uninstall targets that reference any installation paths. I didn't need to add `$(DESTDIR)` to the `uninstall` command for the sake of package managers, because they don't care how your package is uninstalled. Package managers only install your package while building it so they can copy the specified products from the temporary install directory, which they then delete entirely after the package is created. Package managers like RPM use their own rules for removing products from a system, and these rules are based on package manager databases, not your build system.

For the sake of symmetry and to be complete, it doesn't hurt to add `$(DESTDIR)` to `uninstall`. Besides, we need it to be complete for the sake of the `distcheck` target, which we'll now modify to take advantage of our staged installation functionality:

Makefile

```
...
distcheck: $(distdir).tar.gz
        gzip -cd $+ | tar xvf -
        $(MAKE) -C $(distdir) all check
        $(MAKE) -C $(distdir) DESTDIR=\
         $${PWD}/$(distdir)/_inst install uninstall
        $(MAKE) -C $(distdir) clean
        rm -rf $(distdir)
        @echo "*** Package $(distdir).tar.gz\
          ready for distribution."
...
```

Changing the `prefix` variable to the `DESTDIR` variable in the second `$(MAKE)` line above allows us to test a complete install directory hierarchy properly, as we'll see shortly here.

At this point, an RPM spec file (for example) could provide the following text as the installation commands for the jupiter package:

```
%install
make prefix=/usr DESTDIR=%BUILDROOT install
```

But don't worry about package manager file formats. Just focus on providing staged installation functionality through the DESTDIR variable.

You may be wondering why this functionality could not be provided by the `prefix` variable. Well, for one thing, not every path in a system-level installation is defined relative to the `prefix` variable. The system configuration directory (`sysconfdir`), for instance, is often defined simply as `/etc` by packagers. Defining `prefix` to anything other than `/` will have little effect on `sysconfdir` during staged installation, unless a build system uses `$(DESTDIR)$(sysconfdir)` to reference the system configuration directory. Other reasons for this will become more clear as we talk about project configuration later in this chapter.

Build versus installation prefix overrides

At this point, I'd like to digress slightly for just a moment to explain an illusive (or at least non-obvious) concept regarding the prefix and other path variables defined by the GNU Coding Standards document.

In the preceeding examples, I've always used prefix overrides on the `make install` command line, like this:

```
$ make prefix=/usr install

...
```

The question I wish to address is: What's the difference between using a prefix override for `make all` and `make install`? In our small sample makefiles, we've managed to avoid using prefixes in any targets not related to installation, so it may not be clear at this point that a prefix is *ever* useful during the build stages.

One key use of prefix variables during the build stage is to substitute paths into source code at compile time, in this manner:

```
main.o : main.c

        gcc -DCFGDIR=\"$(sysconfdir)\" -o $@ $+
```

In this example, I'm defining a C preprocessor variable called CFGDIR on the compiler command line for use by `main.c`. Presumably, there's some code in `main.c` that looks like this:

```
#ifndef CFGDIR

# define CFGDIR "/etc"

#endif



char cfgdir[FILENAME_MAX] = CFGDIR;
```

Later in the code, the C global variable "`cfgdir`" might be used to access the application's configuration file.

Okay, with that background then, would you ever want to use *different* prefix variable overrides on the build and installation command lines? Sure--Linux distro packagers do this all the time in RPM spec files. During the build stage, the actual run-time directories are hard-coded into the executable by using a command like this:

```
%build
```

```
%setup

./configure prefix=/usr sysconfdir=/etc

make
```

The RPM build process installs these executables into a stage directory, so it can copy them out. The corresponding installation command looks like this:

```
%install

rm -rf %BUILDROOT%

make DESTDIR=%BUILDROOT% install
```

I mentioned the `DESTDIR` variable previously as a tool used by packagers for staged installation. This has the same effect as using:

```
%install

rm -rf %BUILDROOT%

make prefix=%BUILDROOT%/usr \

     sysconfdir=%BUILDROOT%/etc install
```

The key take-away point here is this: Never recompile from an `install` target in your makefiles. Otherwise your users won't be able to access your staged installation features when using prefix overrides.

Another reason for this is to allow the user to install into a grouped location, and then create links to the actual files in the proper locations. Some people like to do this, especially when they are testing out a package, and want to keep track of all of its components. For example, some Linux distributions provide a way of installing multiple versions of some common packages. Java is a great example here. To support using multiple versions or brands (perhaps Sun Java vs IBM Java), the Linux distribution provides a script set called the "alternatives" scripts, which allows a user (running as root) to swap all of the links in the various system directories from one grouped installation to another. Thus, both sets of files may be installed in different auxiliary locations, but links in the true installation locations can be changed to refer to each group at different times.

One final point about this issue. If you're installing into a system directory hierarchy, you'll need root permissions. Often people run `make install` like this:

```
$ sudo make install

...
```

If your `install` target depends on your build targets, and you've neglected to build beforehand, then `make` will happily build your program before installing it, but the local copies will all be owned by `root`. Just an inconvenience, but easily avoided by having `make install' fail for lack of things to install, rather than simply jump right into a build while running as root.

Standard user variables

There's one more topic I'd like to cover before we move on to configuration. The GNU Coding Standards document defines a set of variables that are sort of sacred to the user. That is, these variables should be used by a GNU build system, but never modified by a GNU build system. These are called "user variables", and they include the following for C and C++ programs:

```
CC          - the C compiler

CFLAGS      - C compiler flags

CXX         - the C++ compiler

CXXFLAGS    - C++ compiler flags

LDFLAGS     - linker flags

CPPFLAGS    - C preprocessor flags

...
```

This list is by no means comprehensive, and ironically, there isn't a comprehensive list to be found in the GCS document. Interestingly, most of these user variables come from the documentation for the `make` utility. You can find a fairly complete list of program name and flag variables in section 10.3 of the GNU make manual. The reason for this is that these variables are used in the built-in rules of the make utility.

For our purposes, these few are sufficient, but for a more complex makefile, you should become familiar with the larger list so that you can use them as the occasion arises. To use these in our makefiles, we'll just replace "gcc" with `$(CC)`, and then set `CC` to the gcc compiler at the top of the makefile. We'll do the same for `CFLAGS` and `CPPFLAGS`, although this last one will contain nothing by default:

`src/Makefile`

```
...

CC     = gcc

CFLAGS = -g -O2

...

jupiter: main.c

        $(CC) $(CFLAGS) $(CPPFLAGS) -o $@ $+
```

```
...
```

The reason this works is that the make utility allows such variables to be overridden by options on the command line. `Make` command-line variable assignments always override values set in the makefiles themselves. Thus, to change the compiler and set some compiler flags, a user need simply type:

```
$ make CC=gcc3 CFLAGS='-g -O0' CPPFLAGS=-dtest
```

In this case, our user has decided to use gcc version 3 instead of 4, and to disable optimization and leave the debugging symbols in place. She's also decided to enable the "test" option through the use of a preprocessor definition. Note that these variables are set on the make command line. This apparently equivalent syntax will not work as expected:

```
$ CC=gcc3 CFLAGS='-g -O0' CPPFLAGS=-dtest make
```

The reason for this is that we're merely setting environment variables in the local environment passed to the make utility by the shell. Remember that environment variables do not automatically override those set in the makefile. To get the functionality we want, we could use a little GNU make-specific syntax in our makefile:

```
CC     ?= gcc

CFLAGS ?= -g -O2
```

The "`?=`" operation is a GNU Make-specific operator, which will only set the variable in the makefile if it hasn't already been set elsewhere. This means we can now override these particular variable settings by setting them in the environment. But don't forget that this will only work in GNU Make.

Configuring your package

The GNU Coding Standards document describes the configuration process in section 7.1, "How Configuration Should Work". Up to this point, we've been able to do about everything we've wanted to do with the jupiter project using only makefiles. You might be wondering at this point what configuration is actually for! The opening paragraphs of Section 7.1 state:

Each GNU distribution should come with a shell script named `configure`. This script is given arguments which describe the kind of machine and system you want to compile the program for.
The `configure` script must record the configuration options so that they affect compilation.
One way to do this is to make a link from a standard name such as `config.h` to the proper configuration file for the chosen system. If you use this technique, the distribution should not contain a file named `config.h`. This is so that people won't be able to build the program without configuring it first.

Another thing that `configure` can do is to edit the makefiles. If you do this, the distribution should not contain a file named `Makefile`. Instead, it should include a file `Makefile.in` which contains the input used for editing. Once again, this is so that people won't be able to build the program without configuring it first.
So then, the primary tasks of a typical configure script are to:

- generate files from templates containing replacement variables,
- generate a C language header file (often called `config.h`) for inclusion by project source code,
- set user options for a particular `make` environment--such as debug flags, etc.,
- set various package options as environment variables,
- and test for the existance of tools, libraries, and header files.

For complex projects, `configure` scripts often generate the project makefile(s) from one or more templates maintained by project developers. A makefile template contains configuration variables in an easily recognized (and substituted) format. The `configure` script replaces these variables with values determined during configuration--either from command line options specified by the user, or from a thorough analysis of the platform environment. Often this analysis entails such things as checking for the existence of certain system or package include files and libraries, searching various file system paths for required utilities and tools, and even running small programs designed to indicate the feature set of the shell, C compiler, or desired libraries.
The tool of choice here for variable replacement has, in the past, been the `sed` stream editor. A simple `sed` command can replace all of the configuration variables in a makefile template in a single pass through the file. In the latest version of Autoconf (2.62, as of this writing) prefers `awk` to `sed` for this process. The `awk` utility is *almost* as pervasive as `sed` these days, and it much more powerful with respect to the operations it can perform on a stream of data. For the purposes of the jupiter project, either one of these tools would suffice.

Summary

At this point, we've created a complete project build system by hand--with one important exception. We haven't designed a `configure` script according to the design criteria specified in the GNU Coding Standards document that works with this build system. We could do this, but it would take a dozen more pages of text to build one that even comes close to conforming to these specifications.
There are yet a few key build system features related specifically to the makefiles that are indicated as being desirable by the GNU Coding Standards. Among these is the concept of VPATH building. This is an important feature that can only be properly illustrated by actually writing a `configure` script that works as specified by the GNU Coding Standards.
Rather than spend this time and effort, I'd like to simply move on to a discussion of Autoconf in Chapter 3, which will allow us to build one of these `configure` scripts in as little as two or three lines of code, as you'll see in the opening paragraphs of that chapter. With that step behind us, it will be trival to add VPATH building, and other features to the jupiter project.

Source archive

Download the attached source archive for the original sources associated with this chapter.

‹ Chapter 1: A brief introduction to the GNU Autotools up Chapter 3: Configuring your project with Autoconf ›

Chapter 3: Configuring your project with Autoconf

Tue, 2008-06-10 22:11 -- John Calcote

We should all be very grateful to David MacKenzie for having the foresight to--metaphorically speaking--stop and sharpen the ax. Otherwise we'd still be writing (copying) and maintaining long, complex hand-coded `configure` scripts today.
This chapter has downloads!
Before Automake, Autoconf was used alone, and many legacy open source projects have never really made the transition to the full Autotools suite. As a result, it would not be uncommon to find an open source project containing a file called `configure.in` (the older naming convention used by Autoconf) and hand-written `Makefile.in` templates.

Configure scripts, the Autoconf way

It's instructive for this and other reasons that will become clear shortly, to spend some time just focusing on the use of Autoconf alone. Exploring in this manner can provide a fair amount of insight into the operation of Autoconf by exposing aspects of this tool that are often hidden by Automake and other add-on tools.

The input to Autoconf is ... (drum roll please) ... shell script. Man, what an anti-climax! Okay, so it's not pure shell script. That is, it's shell script with macros, plus a bunch of macro definition files--both those that ship with an Autoconf distribution, as well as those that you or I write. The macro language used is called M4. ("M-what?!", you ask?) The M4 utility is a general purpose macro language processor that was originally written by none other than Brian Kernighan and Dennis Ritchie in 1977. (The name M4 means "m plus 4 more letters" or the word "Macro" - cute, huh? As a point of interest, this naming convention is a fairly common practice in some software engineering domains. For example, the term *internationalization* is often abrieviated *i18n*, and the term *localization* is sometimes replaced with *l10n*, for the sake of brevity. The use of the term *m4* here is no-doubt a play on this concept.)
Some form of the M4 macro language processor is found on *every* Unix and Linux variant (as well as other systems) in use today. In fact, this proliferance is the primary reason for its use in Autoconf. The design goals of Autoconf included primarily that it should run on all systems without the addition of complex tool chains and utility sets. Autoconf depends on the existence of relatively few tools, including `m4`, `sed` and now in version 2.62, the `awk` utility. Most of the Autotools (Autoconf being the exception) rely on the existence of a perl processor, as well.
NOTE: Do not confuse the requirements of the Autotools with the requirements of the scripts and makefiles generated by them. The Autotools are maintainer tools, while the resulting scripts and makefiles are end-user tools. We can reasonably expect a higher level of installed functionality on development systems than we can on end-user systems. Nevertheless, the Autotools design goals still include a reliance only on a minimal set of pre-installed functionality, much of which is part of a default installation.
While it's true that `configure.ac` is written in shell script sprinkled with M4 syntax, the proper use of the M4 macro processor is the subject of Chapter 7. Because I want to stick to Autoconf in this chapter, I'll gloss over some key concepts related to M4, which I'll cover in more detail in Chapter 7. This chapter is designed to help you understand Autoconf concepts, however, so I will cover minor aspects of M4 as it makes sense to do so.

The smallest `configure.ac` file

The simplest possible `configure.ac` file has just two lines:

```
$ cat configure.ac

AC_INIT([jupiter], [1.0])

AC_OUTPUT

$
```

NOTE: This chapter builds on the Jupiter project begun in Chapter 2.

To those new to Autoconf, these two lines appear to be a couple of function calls, perhaps in the syntax of some obscure computer language. Don't let this appearance throw you--these are M4 macro expansions. The macros are defined in files distributed with Autoconf. The definition of `AC_INIT`, for example,W is found in `$PREFIX/share/autoconf/autoconf/general.m4`, while `AC_OUTPUT` is defined in `status.m4`, in the same directory.

M4 macros are similar in many ways to macros defined in C language source files for the C preprocessor, which is also a text replacement tool. This isn't surprising, given that both M4 and `cpp` were originally designed by Kernighan and Ritchie.

The square brackets around the parameters are used by Autoconf as a quoting mechanism. Such quotes are only really necessary in cases where the context of the macro call could cause an ambiguity that the macro processor may resolve incorrectly (usually without telling you). We'll discuss M4 quoting in much more detail in Chapter 7. For now, just use Autoconf quotes ([and]) around every argument to ensure that the expected macro expansions are generated.

As with `cpp` macros, M4 macros may or may not take parameters. And (also as with `cpp`) when they do, then a set of parentheses must be used when passing the arguments. In both M4 and `cpp`, the opening parenthesis must immediately follow the macro name, with no intervening white space. When they don't accept parameters, the parentheses are simply omitted. Unlike `cpp`, M4 has the ability to specify *optional* parameters, in which case, you may omit the parentheses if you choose not to pass a parameter.

The result of passing this `configure.ac` file through Autoconf is essentially the same file (now called `configure`), only with these two macros fully expanded.

Now, if you've been programming in C for many years, as I have, then you've no doubt run across a few C preprocessor macros from the dark regions of the lower realm. I'm talking about those truly evil `cpp` macros that expand into one or two pages of C code! You know the ones I'm talking about--they should really have been written as C functions, but the author was overly worried about performance!

Well baby, you ain't seen *nothin'* yet! These two M4 macros expand into a file containing over 2200 lines of Bourne shell script that's over 60K bytes in size! Interestingly, you wouldn't really know this by looking at their definitions. They're both fairly short--only a dozen or two lines each. The reason for this apparent disparity is simple--they're written in a modular fashion, each macro expanding several others, which in turn expand several others, and so on.

Executing Autoconf

Running Autoconf couldn't be simpler. Just execute `autoconf` in the same directory as your `configure.ac` file. While I *could* do this for each example in this chapter, I'm going to use the `autoREconf` (capitalization added for emphasis) command instead of the `autoconf` command. The reason for this is that running `autoreconf` has exactly the same effect as running `autoconf`, except that `autoreconf` will also do "the right thing" when you start adding Automake and Libtool functionality to your build system.

`autoreconf` is the recommended method for executing the Autotools tool chain, and it's smart enough to only execute the tools that you need, in the order that you need them, and with the options that you need (with one exception that I'll mention here shortly).

```
$ autoreconf

$ ls -lp

autom4te.cache/

configure

configure.ac

$
```

First, notice that `autoreconf` operates at exactly the same level of verbosity as the tools it runs. By default, zero. If you want to see something happening, use the `-v` or `--verbose` option. If you want `autoreconf` to run the other Autotools in verbose mode, add `-vv` to the command line. (You may also pass `--verbose --verbose`, but this syntax seems a bit... verbose to me--sorry, I couldn't resist!)

First, notice that Autoconf creates a directory called `autom4te.cache`. This is the `autom4te` (pronounced "automate") cache directory. This cache is used to speed up access to `configure.ac` by successive executions of utilities in the Autotools tool chain. I'll cover `autom4te` in greater detail in Chapter 9, where I'll show you how to write your own Autoconf macros that are "environmentally friendly".

Executing `configure`

If you recall from the last section of Chapter 2, the GNU Coding Standards document indicates that `configure` should generate a script called `config.status`, whose job it is to generate files from templates. Well, this is exactly the sort of functionality found in an Autoconf-generated `configure` script. An Autoconf-generated `configure` script has two primary tasks:

- perform requested checks
- generate, and then call `config.status`

The results of all of the checks performed by the `configure` script are written, as environment variable settings to the top of `config.status`, which uses the values in these environment variables as replacement text for Autoconf substitution variables it finds in template files (`Makefile.in`, `config.h.in`, etc).

When you execute `configure`, it tells you that it's creating the `config.status` file. In fact, it also creates a log file called `config.log` that has several important attributes:

```
$ ./configure

configure: creating ./config.status

$

$ ls -lp

autom4te.cache/
```

```
config.log

config.status

configure

configure.ac

$
```

The `config.log` file contains the following information:

- the command line used to invoke `configure` (very handy!)
- information about the platform on which `configure` was executed
- information about the core tests executed by `configure`
- the line number in `configure` at which `config.status` is generated and then called

At this point in the log file, `config.status` takes over generating log information--it adds the command line used to invoke `config.status`. After `config.status` generates all of the files from their templates, it then exits, returning control to `configure`, which then adds the following information to the log:

- the cache variables used by `config.status` to perform its tasks
- the list of output variables that may be replaced in templates
- the exit code returned by `configure` to the shell

This information is invaluable when debugging a `configure` script and its associated `configure.ac` file.

Executing `config.status`

Now that you know how `configure` works, you can probably see that there might be times when you'd be tempted to simply execute `config.status` yourself, rather than going to all the trouble of having `configure` perform all those time-consuming checks first. And right you'd be. This was exactly the intent of the Autoconf designers--and the authors of the GNU Coding Standards, by whom these design goals were originally conceived.

There are in fact, times when you'd just like to manually regenerate all of your output files from their corresponding templates. But, far more importantly, `config.status` can be used by your makefiles to regenerate themselves individually from their templates, when `make` determines that something in a template file has changed.

Rather than call `configure` to perform needless checks (your environment hasn't changed, has it? Just your template files), your makefiles should be written in a way that ensures that output files are dependent on their templates. If a template file changes (because, for example, you modified one of your `Makefile.in` templates), then `make` calls `config.status` to regenerate this file. Once the `Makefile` is regenerated, then `make` re-executes the original `make` command line--basically, it restarts itself. This is actually a feature of the `make` utility.

Let's take a look at the relevant portion of just such a `Makefile.in` template:

```
Makefile: Makefile.in config.status

        ./config.status Makefile
```

Another interesting bit of `make` functionality is that it always looks for a rule with a target named "`Makefile`". Such a rule allows `make` to regenerate the source makefile from its template, in the event that the template changes. It does this *before* executing either the user's specified targets, or the default target, if none was given.

This example indicates that `Makefile` is dependent on `Makefile.in`. Note that `Makefile` is also dependent on `config.status`. After all, if `config.status` is regenerated by the `configure` script, then it may generate a makefile differently--perhaps something in the compilation environment changed, such as when a new package is added to the system, so that `configure` can now find libraries and headers not previously found. In this case, Autoconf substitution variables may have different values. Thus, `Makefile` should be regenerated if either `Makefile.in` or `config.status` changes.

Since `config.status` is itself a generated file, it stands to reason that this line of thinking can be carried to the `configure` script as well. Expanding on the previous example:

```
Makefile: Makefile.in config.status
        ./config.status $@


config.status: configure
        ./config.status --recheck
```

Since `config.status` is a dependency of the `Makefile` rule, then `make` will check for a rule whose target is `config.status` and run its commands if the dependencies of `config.status` (`configure`) are newer than `config.status`.

Adding some real functionality

Well, it's about time we move forward and put some true functionality into this `configure.ac` file. I've danced around the topic of having `config.status` generate a makefile up to this point. Here's the code to actually make this happen in `configure.ac`. It constitutes a single additional macro expansion between the original two lines:

```
$ cat configure.ac
AC_INIT([jupiter], [1.0])
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
$
```

This code assumes that I have templates for `Makefile` and `src/Makefile`, called `Makefile.in` and `src/Makefile.in`, respectively. These files look exactly like their `Makefile` counterparts, with one exception: Any text that I want Autoconf to replace should be marked as Autoconf substitution variables, using the `@VARIABLE@` syntax.

To create these files, I've merely renamed the existing makefiles to `Makefile.in` within the top-level and `src` directories. By the way, this is a common practice when "autoconfiscating" a project. Next, I added a few Autoconf substitution variables to replace my orignal default values. In fact, at the top of this file, I've added the special Autoconf substitution variable, `@configure_input@` after a makefile comment HASH mark. This comment line will become the following text line in the generated `Makefile`:

```
# "Makefile.  Generated from Makefile.in by conf...
```

I've also added the makefile regeneration rules (from the examples above) to each of these templates, with slight file path differences in each file to account for their different positions relative to `config.status` and `configure`:

`Makefile.in`

```
# @configure_input@


# Package-related substitution variables
package        = @PACKAGE_NAME@
version        = @PACKAGE_VERSION@
tarname        = @PACKAGE_TARNAME@
distdir        = $(tarname)-$(version)


# Prefix-related substitution variables
prefix         = @prefix@
exec_prefix    = @exec_prefix@
bindir         = @bindir@
...
$(distdir):
	mkdir -p $(distdir)/src
	cp configure $(distdir)
	cp Makefile.in $(distdir)
	cp src/Makefile.in $(distdir)/src
```

```
        cp src/main.c $(distdir)/src

distcheck: $(distdir).tar.gz
        gzip -cd $+ | tar xvf -
        cd $(distdir); ./configure
        $(MAKE) -C $(distdir) all check
        $(MAKE) -C $(distdir) \
         DESTDIR=$${PWD}/$(distdir)/_inst \
         install uninstall
        $(MAKE) -C $(distdir) clean
        rm -rf $(distdir)
        @echo "*** Package $(distdir).tar.gz is\
         ready for distribution."

Makefile: Makefile.in config.status
        ./config.status $@

config.status: configure
        ./config.status --recheck
...
```

src/Makefile.in

```
# @configure_input@

# Package-related substitution variables
package        = @PACKAGE_NAME@
```

```
version         = @PACKAGE_VERSION@

tarname         = @PACKAGE_TARNAME@

distdir         = $(tarname)-$(version)



# Prefix-related substitution variables

prefix          = @prefix@

exec_prefix     = @exec_prefix@

bindir          = @bindir@

...

Makefile: Makefile.in ../config.status

        cd .. && ./config.status $@



../config.status: ../configure

        cd .. && ./config.status --recheck

...
```

I've removed the `export` statement in the top-level `Makefile.in`, and added a copy of all of the substitution variables into `src/Makefile.in`. Since `config.status` is generating both of these files, I can reap excellent benefits by substituting everything into both files. The primary advantage of doing this is that I can now run `make` in any sub-directory, and not be concerned about environment variables that would have been passed down by a higher-level makefile.

Finally, I've changed the distribution targets a bit. Rather than distribute the makefiles, I now want to distribute the `Makefile.in` templates, as well as the `configure` script. In addition, the `distcheck` target needed to be enhanced such that it runs the `configure` script before attempting to run `make`.

Generating files from templates

I'm now generating makefiles from `Makefile.in` templates. The fact is, however, that *any* (white space delimited) file listed in `AC_CONFIG_FILES` will be generated from a file of the same name with a "`.in`" extension, found in the same directory. The "`.in`" extension is the default template naming pattern for `AC_CONFIG_FILES`, but this default behavior may be overridden, if you wish. I'll get into the details shortly.

Autoconf generates `sed` or `awk` expressions into the resulting `configure` script, which then copies them into the `config.status` script. The `config.status` script uses these tools to perform this simple string replacement.

Both `sed` and `awk` are text processing tools that operate on file streams. The advantage of a stream editor (the name "`sed`" is actually a contraction of the phrase "stream editor") is that it replaces text patterns in a byte stream. Thus, both `sed` and `awk` can operate on huge files, because they don't need to load the entire input file into memory in order to process it. The expression list passed to `sed` or `awk` by `config.status` is built by Autoconf from a list of variables defined by various macros, many of which I'll cover in greater detail in this chapter.

The important thing to notice here is that the Autoconf variables are the *only* items replaced in `Makefile.in` while generating the makefile. The reason this is important to understand is that it helps you to realize the flexibility you have when allowing Autoconf to generate a file from a template. This flexibility will become more apparent as I get into various use cases for the pre-defined Autoconf macros, and later in Chapter 9 when I delve into the topic of writing your own Autoconf macros.

At this point, I've created a basic `configure.ac` file, and I can indeed run `autoreconf`, followed by the generated `configure` script, and then `make` to build the Jupiter project.

The idea that I want to promote at this point is that *this simple three-line `configure.ac` file generates a `configure` script that is fully functional, according to the definition of a `configure` script given in Chapter 7 of the the GNU Coding Standards document*.

The resulting `configure` script runs various system checks and generates a `config.status` file, which can replace a fair number of substitution variables in a set of specified template files in a build system. That's a lot of stuff for three lines of code. (You'll recall my comments in the introduction to this book about C++ doing a lot for you with just a few lines of code?)

Adding VPATH build functionality

Okay, you may recall at the end of Chapter 2, I mentioned that I hadn't yet covered a key concept--that of `VPATH` builds. A `VPATH` build is a way of using a particular makefile construct (`VPATH`) to configure and build a project in a directory other than the source directory. Why is this important? Well, for several reasons. You may need to:

- maintain a separate debug configuration,
- test different configurations, side by side,
- keep a clean source directory for patch diffs after local modifications,
- or build from a read-only source directory.

These are all great reasons, but won't I have to change my entire build system to support this type of remote build? As it turns out, it's quite simple using the `make` utility's `VPATH` statement. `VPATH` is short for "virtual path", meaning "virtual search path". A `VPATH` statement contains a colon-separated list of places to look for dependencies, when they can't be found relative to the current directory:

```
VPATH = some/path:some/other/path:yet/another/path

jupiter : main.c
        gcc ...
```

In this (contrived) example, if `make` can't find `main.c` in the current directory while processing the rule, it will look for `some/path/main.c`, and then for `some/other/path/main.c`, and finally for `yet/another/path/main.c`, before finally giving up in dispair--okay, perhaps only with an error message about not knowing how to make `main.c`.

"Nice feature!", you say? Nicer than you think, because with just a few simple modifications, I can now completely support remote builds in my jupiter project build system:

`Makefile.in`

```
...
# VPATH-related substitution variables
srcdir         = @srcdir@
VPATH          = @srcdir@
...
$(distdir):
        mkdir -p $(distdir)/src
        cp $(srcdir)/configure $(distdir)
        cp $(srcdir)/Makefile.in $(distdir)
        cp $(srcdir)/src/Makefile.in $(distdir)/src
        cp $(srcdir)/src/main.c $(distdir)/src
...
```

`src/Makefile.in`

```
...
# VPATH-related substitution variables
srcdir         = @srcdir@
VPATH          = @srcdir@
...
jupiter: main.c
        gcc -g -O0 -o $@ $(srcdir)/main.c
...
```

That's it. Really. When `config.status` generates a file, it replaces an Autoconf substitution variable called `@srcdir@` with the relative path to the template's source directory. Each makefile will get a different value for `@srcdir@`, depending on the relative location of its template.

The rules then for supporting `VPATH` builds in your make system are as follows:

- Set a `make` variable, `srcdir` to the `@srcdir@` substitution variable.
- Set VPATH to `@srcdir@` also--don't use `$(srcdir)` because some older versions of `make` don't do variable substitution within the value of `VPATH`.
- Prefix all file dependencies used *in commands* with `$(srcdir)/`.

  If the source directory is the same as the build directory, then the `@srcdir@` substitution variable degenerates to "`.`", so all of these "`$(srcdir)/`" prefixes degenerate to "`./`", which is just so much harmless baggage.

  A quick example is the easiest way to show you how this works. Now that Jupiter is fully functional with respect to `VPATH` builds, let's just give it a try. Start in the `jupiter` project directory, create a subdirectory called "`build`", and then change into that directory. Now run `configure` using a relative path, and then list the current directory contents:

```
$ mkdir build

$ cd build

$ ../configure

configure: creating ./config.status

config.status: creating Makefile

config.status: creating src/Makefile

$ ls -1p

config.log

config.status

Makefile

src/

...
```

  The entire build system seems to have been constructed by `configure` and `config.status` within the `build` sub-directory, just as it should be. What's more, it actually works:

```
...

$ make

make -C src all

make[1]: Entering directory `../prj/jupiter/bui...

gcc -g -O2 -o jupiter ../../src/main.c
```

```
make[1]: Leaving directory `../prj/jupiter/bui...

$ ls -1p src

jupiter

Makefile
```

`VPATH` builds work, not just from sub-directories of the project directory, but from anywhere you can access the project directory, using either a relative or an absolute path. This is just one more thing that Autoconf does for you in Autoconf-generated `configure` scripts. Just imagine managing proper relative paths to source directories in your own hand-coded `configure` scripts!

Let's take a breather

At this point, I'd like you to stop and consider what you've seen so far: I've shown you a mostly complete build system that includes most of the features outlined in the GNU Coding Standards document. The features of the Jupiter project's make system are all fairly self-contained, and reasonably simple to grasp. The most difficult feature to implement by hand is the `configure` script. In fact, writing a `configure` script by hand is so labor intensive relative to the simplicity of the Autoconf version that I just skipped over the hand-coded version entirely in Chapter 2.

If you've been one to complain about Autoconf in the past, I'd like you to consider what you have to complain about now. You now know how to get very feature-rich configuration functionality *in just three lines of code.* Given what you know now about how `configure` scripts are meant to work, can you see the value in Autoconf?

Most people never have trouble with that portion of Autoconf that I've covered up to this point. The trouble is that most people don't create their build systems in the manner I've just shown you. They try to copy the build system of another project, and then tweak it to make it work in their own project. Later when they start a new project, they do the same thing again. Are they going to run into problems? Sure--the "stuff" they're copying was often never meant to be used the way they're trying to use it.

I've seen projects in my experience whose `configure.ac` file contained junk that had nothing to do with the project to which it belonged. These left-over bits came from the previous project, from which `configure.ac` was copied. But the maintainer didn't know enough about Autoconf to remove the cruft. With the Autotools, it's better to start small, and add what you need, than to start with a full-featured build system, and try to pare it down to size.

Well, I'm sure you're feeling like there's a lot more learn about Autoconf. And you're right, but what additional Autoconf macros are appropriate for the Jupiter project?

An even quicker start with `autoscan`

The simplest way to create a (mostly) complete `configure.ac` file is to run the `autoscan` utility, which, if you remember from Chapter 1, is part of the Autoconf package. First, I'll clean up the droppings from my earlier experiments, and then run the `autoscan` utility in the `jupiter` directory. Note here that I'm NOT deleting my original `configure.ac` file - I'll just let `autoscan` tell me what's wrong with it. In less than a second I'm left with a couple of new files in the top-level directory:

```
$ rm config.* Makefile src/Makefile ...
```

```
$ ls -1p
configure.ac
Makefile.in
src/
$ autoscan
configure.ac: warning: missing AC_CHECK_HEADERS
   ([stdlib.h]) wanted by: src/main.c:2
configure.ac: warning: missing AC_HEADER_STDC
   wanted by: src/main.c:2
configure.ac: warning: missing AC_PROG_CC
   wanted by: src/main.c
configure.ac: warning: missing AC_PROG_INSTALL
   wanted by: Makefile.in:11
$ ls -1p
autom4te.cache/
autoscan.log
configure.ac
configure.scan
Makefile.in
src/
```

NOTE: I've wrapped some of the output lines for the sake of column width during publication.

`autoscan` creates two files called `configure.scan`, and `autoscan.log` from a project directory hierarchy. The project may already be instrumented for Autotools, or not. It doesn't really matter because `autoscan` is decidedly non-destructive. It will never alter any existing files in a project.

`autoscan` generates a warning message for each issue discovered in an existing `configure.ac` file. In this example, `autoscan` noticed that `configure.ac` really should be using the `AC_CHECK_HEADERS`, `AC_HEADER_STDC`, `AC_PROG_CC` and `AC_PROG_INSTALL` macros. It made these assumptions based on scanning my existing `Makefile.in` templates and C source files, as you can see by the comments after each warning statement. You can always see these messages (in even greater detail, in fact) by examining the `autoscan.log` file.

Now let's take a look at the generated `configure.scan` file. `autoscan` has added more text to `configure.scan` than was originally in my `configure.ac` file, so it's probably easier for me to just overwrite `configure.ac` with `configure.scan` and then change the few bits of information that are specific to Jupiter:

```
$ mv configure.scan configure.ac

$ cat configure.ac

#                -*- Autoconf -*-

# Process this file with autoconf to produce ...



AC_PREREQ(2.61)

AC_INIT(FULL-PACKAGE-NAME, VERSION,

        BUG-REPORT-ADDRESS)

AC_CONFIG_SRCDIR([src/main.c])

AC_CONFIG_HEADERS([config.h])



# Checks for programs.

AC_PROG_CC

AC_PROG_INSTALL



# Checks for libraries.



# Checks for header files.

AC_HEADER_STDC

AC_CHECK_HEADERS([stdlib.h])



# Checks for typedefs, structures, and compiler ...
```

```
# Checks for library functions.


AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

NOTE: The contents of your `configure.ac` file may differ slightly from mine, depending on the version of Autoconf that you have installed. I have version 2.62 of GNU Autoconf installed on my system (the latest, as of this writing), but if your version of `autoscan` is older (or newer), you may see some minor differences.

I'll then edit the file and change the `AC_INIT` macro to reflect the Jupiter project parameters:

```
$ head configure.ac
#                -*- Autoconf -*-
# Process this file with autoconf to produce ...


AC_PREREQ([2.61])
AC_INIT([jupiter], [1.0], [bugs@jupiter.org])
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADERS([config.h])
$
```

The `autoscan` utility really does a lot of the work for you. The GNU Autoconf manual states that you should hand-tailor this file to your project before using it. This is true, but there are only a few key issues to worry about (besides those related to the `AC_INIT` macro). I'll cover each of these issues in turn, starting at the top of the file.

Trying out `configure`

I like to experiment, so the first thing I'd do at this point would be to try to run `autoreconf` on this new `configure.ac`. and then try to run the generated `configure` script to see what happens. If `autoscan` is all it's cracked up to be, then the resulting `configure` script should generate some makefiles for me:

```
$ autoreconf
$ ./configure
```

```
checking for gcc... gcc

checking for C compiler default output file name...

checking whether the C compiler works... yes

checking whether we are cross compiling... no

checking for suffix of executables...

checking for suffix of object files... o

checking whether we are using the GNU C compiler...

checking whether gcc accepts -g... yes

checking for gcc option to accept ISO C89...

configure: error: cannot find install-sh or

   install.sh in "." "./.." "./../.."

$
```

Well, we didn't get too far. I mentioned the `install` utility in Chapter 1, and you may have already been aware of it. It appears here that Autoconf is looking for a shell script called `install-sh` or `install.sh`.

Autoconf is all about portability, and unfortunately, the `install` utility is not as portable as we'd like it to be. From one platform to another, critical bits of installation functionality are just different enough to cause problems, so the Autotools provide a shell script called `install-sh` (deprecated name: `install.sh`) that acts as a wrapper around the platform `install` utility. This wrapper script masks important differences between various versions of `install`.

`autoscan` noticed that I used the `install` program in my `src/Makefile.in` template, and generated an expansion of the `AC_PROG_INSTALL` macro into the `configure.scan` file based on this observation. The problem is that the generated `configure` script couldn't find the `install-sh` wrapper script.

This seems to be a minor defect in Autoconf--if Autoconf expects `install-sh` to be in my project directory, then it should have just put it there, right? Well, `autoreconf` has a command line option, `--install`, which is supposed to install missing files like this for me. I'll give it a try. Here's a before-and-after picture of my directory structure:

```
$ ls -1p

autoscan.log

configure.ac

Makefile.in

src/
```

```
$ autoreconf --install

$ ls -1p

autom4te.cache/

autoscan.log

config.h.in

configure

configure.ac

Makefile.in

src/
```

Hmmm. It didn't seem to work, as there's no `install-sh` file in the directory after running `autoreconf --install`. This is, in my opinion, a defect in both `autoreconf` and `autoconf`. You see, when `autoreconf` is used with the `--install` command-line option, it *should* install all auxilliary files required by all Autoconf macros used in `configure.ac`. The trouble is, this auxilliary-file-installation functionality is actually a part of Automake, not Autoconf. So when you use `--install` on the `autoreconf` command-line, it passes tool-specific install-missing-files options down to each of the tools that it calls. This technique would have worked just fine, except that Autoconf doesn't *provide* an option to install any missing files.

Worse still, the GNU Autoconf manual tells you in Section 5.2.1, under `AC_PROG_INSTALL`, that "Autoconf comes with a copy of `install-sh` that you can use." But this is a lie. In fact, it's Automake and Libtool that come with copies of `install-sh`, not Autoconf.

I could just copy `install-sh` from the Automake installation directory (`PREFIX/share/automake...`), but I'll just try running `automake --add-missing --copy` instead. The Automake `--install-missing` option copies in the missing required utility scripts, and the `--copy` option indicates that true copies should be made. Without the `--copy` option, `automake` would actually just create links to these files where they're installed (usually `/usr/(local/)share/automake-1.10`):

```
$ automake --add-missing --copy

configure.ac: no proper invocation of AM_INIT_...

configure.ac: You should verify that configure...

configure.ac: that aclocal.m4 is present in th...

configure.ac: and that aclocal.m4 was recently...

configure.ac:11: installing `./install-sh'

automake: no `Makefile.am' found for any confi...
```

Ignoring the warnings indicating that I've not yet configured my project properly for Automake, I can now see that `install-sh` was copied into my project root directory:

```
$ ls -1p

autom4te.cache/

autoscan.log

configure.ac

configure.scan

install-sh

Makefile.in

src/
```

So why didn't `autoreconf --install` do this for me? Isn't it supposed to run all the programs that it needed to, based on my `configure.ac` script? As it happens, it was exactly *because* my project was not configured for Automake, that `autoreconf` failed to run `automake --add-missing --copy`. Autoreconf saw no reason to run `automake` because `configure.ac` doesn't contain the requisite macros for initializing Automake. And therein lies the defect. First, Autoconf should ship with `install-sh`, since it provides a macro that requires it, and because `autoscan` adds that macro based on the contents of a `Makefile.in` template. In addition, Autoconf should provide an "add-missing" command-line option, and `autoreconf` should use it when called with the `--install` option. This is most likely an example of the "work-in-progress" nature of the Autotools.

But, taking a step backward for a moment. There is another obvious solution to this problem. The install.sh script is not really required by any code generated by Autoconf. How could it be. Autoconf doesn't generate any makefile constructs, it only substitutes variables into your `makefile.in` templates. Thus, there's really no reason for Autoconf to complain about a missing `install-sh` script. When I presented this problem on the Autoconf mailing list, I was told several times that autoconf has no business copying `install-sh` into a project directory, thus there is no such functionality accessible from the Autoconf command line. If that is indeed the case, then Autoconf has no business complaining about the missing file. Regardless, something needs to be fixed...

The proverbial `autogen.sh` script

Before `autoreconf` came along, maintainers used a shell script, often called `autogen.sh`, to run all of the Autotools required for their projects in the proper order. The `autogen.sh` script is often fairly sophisticated, but to solve this problem temporarily, I'll just add a simple temporary `autogen.sh` script to the project root directory:

```
$ echo "automake --add-missing --copy

> autoreconf --install" > autogen.sh

chmod 0755 autogen.sh
```

If you don't want to see all the error messages from `automake`, just redirect the `stderr` and `stdout` output to `/dev/null`.
Eventually, we'll be able to get rid of `autogen.sh` file, and just run `autoreconf --install`, but for now, this will solve our missing files problems. Hopefully, you read this section before scratching your head too much over the missing `install-sh` script. I can now run my newly generated `configure` script without errors. I'll cover the details of properly using the `AC_PROG_INSTALL` macro shortly. I'll cover Automake in much greater detail in Chapter 4.

Updating `Makefile.in`

Okay, so how do the additional macros added by `autoscan` affect my build system? Well, I have some new files to consider. For one, the `config.h.in` file is generated for me now by `autoheader`. I can assume that `autoreconf` now executes `autoheader` for me when I run it. Additionally, I have a new file in my project called `install-sh`. Anything provide by, or generated by the Autotools should be copied into the archive directory so that it can be shipped with my release tarballs. So, I should add these two files to the `$(distdir)` target in the top-level `Makefile.in` template. Note that I don't need to install `autogen.sh`, as it's purely a maintainer tool--my users shouldn't ever need to execute it from a tarball distribution:

`Makefile.in`

```
...

$(distdir):

        mkdir -p $(distdir)/src

        cp $(srcdir)/configure $(distdir)

        cp $(srcdir)/config.h.in $(distdir)

        cp $(srcdir)/install-sh $(distdir)

        cp $(srcdir)/Makefile.in $(distdir)

        cp $(srcdir)/src/Makefile.in $(distdir)/src

        cp $(srcdir)/src/main.c $(distdir)/src

...
```

If you're beginning to think that this could become a maintenance nightmare, then you're right. I warned you in Chapter 2 that the `$(distdir)` target was painful to maintain. Luckily the `distcheck` target still exists, and still works as designed. It would have caught this problem, because the distribution build will not work without these additional files, and certainly the `check` target wouldn't work, if the build didn't work. When I discuss Automake in Chapter 4, much of this mess will be cleared up.

Initialization and package information

The first section in my new `configure.ac` file (copied from `configure.scan`) contains Autoconf initialization macros. These are required for all projects. Let's consider each of these macros individually, as they're all pretty important.

AC_PREREQ

The `AC_PREREQ` macro simply defines the lowest version of Autoconf that may be used to successfully process the `configure.ac` script. The manual indicates that `AC_PREREQ` is the only macro that may be used before `AC_INIT`. The reason for this should be obvious--you'd like to be able to ensure you're using a late enough version of Autoconf before you begin processing any other macros, which may be version dependent. As it turns out, `AC_INIT` is not version dependent anyway, so you may place it first, if you're so inclined. I happen to prefer the way `autoscan` generates the file, so I'll leave it alone.

AC_INIT

The `AC_INIT` macro, as its name implies, initializes the Autoconf system. It accepts up to four arguments (`autoscan` only generates a call with the first three), `PACKAGE`, `VERSION`, and optional `BUG-REPORT` and `TARNAME` arguments. The `PACKAGE` argument is intended to be the name of the package. It will end up (in a canonicalized form) as the first part of the name of an Automake-generated release distribution tarball when you run "`make dist`".

In fact, by default, Automake-generated tarballs will be named `TARNAME-VERSION.tar.gz`, but `TARNAME` is set to a canonicalized form of the `PACKAGE` string (lower-cased, with all punctuation converted to underscores), unless you specify `TARNAME` manually, so bear this in mind when you choose your package name and version string. Incidentally, M4 macro arguments, including `PACKAGE` and `VERSION`, are just strings. M4 doesn't attempt to interpret any of the text that it processes.

The optional `BUG-REPORT` argument is usually set to an email address, but it can be any text really. An Autoconf substitution variable called `PACKAGE_BUGREPORT` will be created for it, and that variable will be added to a `config.h.in` template as a C preprocessor string, as well. The intent is that you use the variable in your code (or in template text files anywhere in your project) to present an email address for bug reports at appropriate places--possibly when the user requests help or version information from your application.

While the `VERSION` argument can be anything you like, there are a few free software conventions that will make life a little easier for you if you follow them. The widely used convention is to pass in *major.minor* (eg., 1.2). However, there's nothing that says you can't use *major.minor.revision* if you want, and there's nothing wrong with this approach. None of the resulting `VERSION` macros (Autoconf, shell or `make`) are parsed or analysed anywhere--only used in various places as replacement text, so if you wish, you may even add non-numeric text into this macro, such as `0.15.alpha1`, which is useful occasionally.

Note that the RPM package manager does indeed care what you put in the version string. For the sake of RPM, you may wish to limit the version string text to only alpha-numerics and periods--no dashes or underscores, unfortunately.

Autoconf will generate the substitution variables `PACKAGE_NAME`, `PACKAGE_VERSION`, `PACKAGE_TARNAME`, `PACKAGE_STRING` (a stylized concatenation of the package name and version information), and `PACKAGE_BUGREPORT` from arguments to `AC_INIT`.

AC_CONFIG_SRCDIR

The `AC_CONFIG_SRCDIR` macro is just a sanity check. Its purpose is to ensure that the generated `configure` script knows that the directory on which it is being executed is in fact the correct project directory. The argument can be a relative path to any source file you like - I try to pick one that sort of defines the project. That way, in case I ever decide to reorganize source code, I'm not likely to lose it in a file rename. But it doesn't really matter, because if you do rename the file or move it to some other location some time down the road, you can always change the argument passed to `AC_CONFIG_SRCDIR`. Autoconf will tell you immediately if it can't find this file--after all, that's the purpose of this macro in the first place!

The instantiating macros

Before we dive into the details of `AC_CONFIG_HEADERS`, I'd like to spend a little time on the framework provided by Autoconf. From a high-level perspective, there are four major things happening in `configure.ac`:

- Initialization
- File instantiation
- Check requests
- Generation of the `configure` script

We've pretty much covered initialization--there's not much to it, although there are a few more macros you should be aware of. (Check out the GNU Autoconf manual to see what these are--look up `AC_COPYRIGHT`, for an example.) Now, let's move on to file instantiation.

There are actually four so-called "instantiating macros", which include `AC_CONFIG_FILES`, `AC_CONFIG_HEADERS`, `AC_CONFIG_COMMANDS` and `AC_CONFIG_LINKS`. An instantiating macro is one which defines one or more tags, usually referring to files that are to be translated by the generated `configure` scripts, from a template containing Autoconf substitution variables.

NOTE: You might need to change the name of `AC_CONFIG_HEADER` (singular) to `AC_CONFIG_HEADERS` (plural) in your version of `configure.scan`. This was a defect in `autoscan` that had not been fixed yet in Autoconf version 2.61. I reported the defect and a patch was committed. Version 2.62 works correctly. If your `configure.scan` is generated with a call to `AC_CONFIG_HEADER`, just change it manually. Both macros will work, as the singular version was the older name of this macro, but the older macro is less functional than the newer one.

These four instantiating macros have an interesting signature in common:

```
AC_CONFIG_xxxS([tag ...], [commands], [init-cmds])
```

For each of these four macros, the `tag` argument has the form, `OUT[:INLIST]` where `INLIST` has the form, `IN0[:IN1:...:INn]`. Often, you'll see a call to one of these macros with only a single simple argument, like this:

```
AC_CONFIG_HEADERS([config.h])
```

In this case, `config.h` is the `OUT` portion of the above specification. The default `INLIST` is the `OUT` portion with "`.in`" appended to it. So the above call is exactly equivalent to:

```
AC_CONFIG_HEADERS([config.h:config.h.in])
```

What this means is that `config.status` will contain shell code that will generate `config.h` from `config.h.in`, substituting all Autoconf variables in the process. You may also provide a list of input files to be concatenated, like this:

```
AC_CONFIG_HEADERS([config.h:cfg0:cfg1:cfg2])
```

In this example, `config.status` will generate `config.h` by concatenating `cfg0`, `cfg1` and `cfg2`, after substituting all Autoconf variables. The GNU Autoconf manual calls this entire "`OUT:INLIST`" thing a "tag".

So, what's that all about, anyway? Why not call it a file? Well, the fact is, this parameter's primary purpose is to provide a sort of command-line target name--much like `Makefile` targets. It also *happens* to be used as a file system name, *if the associated macro happens* to generate file system entries, as is the case when calling `AC_CONFIG_HEADERS`, `AC_CONFIG_FILES` and `AC_CONFIG_LINKS`.

But `AC_CONFIG_COMMANDS` doesn't actually generate any files. Rather, it runs arbitrary shell code, as specified by the user in the macro. Thus, rather than name this first parameter after a secondary function (the generation of files), the manual refers to it by its primary purpose - as a command line tag-name that may be specified on the `config.status` command line. Here's an example:

```
./config.status config.h
```

This `config.status` command line will regenerate the `config.h` file based on the macro call to `AC_CONFIG_HEADERS` in `configure.ac`. It will *only* regenerate `config.h`. Now, if you're curious like me, you've already been playing around a little, and have tried typing `./config.status --help` to see what options are available when executing `config.status`. You may have noticed that `config.status` has a help signature like this:

```
$ ./config.status --help

`config.status' instantiates files from templates

according to the current configuration.



Usage: ./config.status [OPTIONS] [FILE]...



  -h, --help       print this help, then exit

...

  --file=FILE[:TEMPLATE]

...

Configuration files:

 Makefile src/Makefile



Configuration headers:
```

```
config.h
```

NOTE: I left out portions of the help display irrelevant to this discussion.

I'd like you to notice a couple of interesting things about this help display. First, `config.status` is designed to give you custom help about this particular project's `config.status` file. It lists "Configuration files" and "Configuration headers" that you may use as tags. Oddly, given the "tag" nomenclature used in the manual so rigorously, the help line still refers to such tags as `[FILE]`s in the "Usage:" line. Regardless, where the usage specifies `[FILE]`s you may use one or more of the listed configuration files, headers, links, or commands displayed below it. In this case, `config.status` will only instantiate those objects. In the case of commands, it will execute the commands specified by the tag passed in the associated expansion of the `AC_CONFIG_COMMANDS` macro.

Each of these macros may be used multiple times in a `configure.ac` script. The results are cumulative. This means that I can use `AC_CONFIG_FILES` as many times as I need to in my `configure.ac` file. Reasons why I may want to use it more than once are not obvious right now, but I'll get to them eventually.

Another noteworthy item here is that there is a `--file` option. Now why would `config.status` allow us to specify files either with or without the `--file=` in front of them? Well, these are actually different usages of the `[FILE]` option, which is why it would make more sense for the usage text to read:

```
$ ./config.status --help

...

Usage: ./config.status [OPTIONS] [TAG]...
```

When `config.status` is called with tag names on the command line, only those tags listed in the help text as available configuration files, headers, links and commands may be used as tags. When you execute `config.status` with the `--file=` option, you're really telling `config.status` to generate a new file *not already associated with any of the calls to instantiating macros in your `configure.ac` script.* The file is generated from a template using configuration options and check results determined by the the last execution of the `configure` script. For example, I could execute `config.status` like this:

```
./config.status --file=extra:extra.in
```

NOTE: The default template name is the file name with a ".in" suffix, so this call could have been made without using the "`:extra.in`" portion of the option.

Let's get back to the instantiating macro signature. The `tag` argument has a complex format, but it also represents multiple tags. Take another look:

```
AC_CONFIG_xxxS([tag ...], [commands], [init-cmds])
```

The elipsis after `tag` indicates there may be more than one, and in fact, this is true. The `tag` argument accepts multiple tag specifications, separated by white space or new-line characters. Often you'll see a call like this:

`configure.ac`

```
...
AC_CONFIG_FILES([Makefile
                 src/Makefile
                 lib/Makefile
                 etc/project.cfg])
...
```

Each entry here is one tag specification, which if fully specified would look like this:

configure.ac

```
...
AC_CONFIG_FILES([Makefile:Makefile.in
                 src/Makefile:src/Makefile.in
                 lib/Makefile:lib/Makefile.in
                 etc/proj.cfg:etc/proj.cfg.in])
...
```

There's still one more point to cover. There are two optional arguments that you'll not often see used in the instantiating macros, `commands` and `init-cmds`. The `commands` argument may be used to specify some arbitrary shell code that should be executed by `config.status` just before the files associated with the tags are generated. You'll not often see this used with the file generating instantiating macros, but in the case of `AC_CONFIG_COMMANDS`, which generates no files by default, you almost always see this arugument used, because a call to this macro is basically useless without it! In this case, the `tag` argument becomes a way of telling `config.status` to execute a set of shell commands.

The `init-cmds` argument is used to initialize shell variables at the top of `config.status` with values available in `configure.ac` and `configure`. It's important to remember that all calls to instantiating macros share a common namespace along with `config.status`, so choose shell variable names carefully.

The old adage about the relative value of a picture vs. an explanation holds true here, so let's try a little experiment. Create a test version of your `configure.ac` file containing only the following lines:

configure.ac

```
AC_INIT(test, 1.0)
```

```
AC_CONFIG_COMMANDS([abc],

                   [echo "Testing $mypkgname"],

                   [mypkgname=$PACKAGE_NAME])

AC_OUTPUT
```

Then execute `autoreconf`, `configure`, and `config.status` in various ways to see what happens:

```
$ autoreconf

$ ./configure

configure: creating ./config.status

config.status: executing abc commands

Testing test

$ ./config.status

config.status: executing abc commands

Testing test

$ ./config.status --help

`config.status' instantiates files from templates

according to the current configuration.



Usage: ./config.status [OPTIONS] [FILE]...

...

Configuration commands:

 abc



Report bugs to <bug-autoconf@gnu.org>.

$ ./config.status abc

config.status: executing abc commands
```

```
Testing test

$
```

As you can see here, executing `configure` caused `config.status` to be executed with no command line options. There are no checks specified in `configure.ac`. so executing `config.status` has nearly the same effect. Querying `config.status` for help indicates that "abc" is a valid tag, and executing `config.status` with that tag simply runs the associated commands.

Okay, enough fooling around. The important points to remember here are:

- Both `configure` and `config.status` may be called individually to perform their individual tasks.
- The `config.status` script generates all files from templates.
- The `configure` script performs all checks and then executes `config.status`.
- `config.status` generates files based on the last set of check results.
- `config.status` may be called to execute file generation or command sets specified by any of the tag names given in any of the instantiating macro calls.
- `config.status` may generate files not associated with any tags specified in `configure.ac`.
- `config.status` can be used to call `configure` with the same set of command line options used in the last execution of `configure`.

AC_CONFIG_HEADERS

As you've no doubt concluded by now, the `AC_CONFIG_HEADERS` macro allows you to specify one or more header files to be generated from template files. You may write multiple template header files yourself, if you wish. The format of a configuration header template is very specific:

```
/* Define as 1 if you have unistd.h. */

#undef HAVE_UNISTD_H
```

Multiple such statements may be placed in your header template. The comments are optional, of course. Let's try another experiment. Create a new `configure.ac` file with the following contents:

configure.ac

```
AC_INIT([test], [1.0])

AC_CONFIG_HEADERS([config.h])

AC_CHECK_HEADERS([unistd.h foobar.h])

AC_OUTPUT
```

Now create a configuration header template file called `config.h.in`, which contains the following two lines:

config.h.in

```
#undef HAVE_UNISTD_H
```

```
#undef HAVE_FOOBAR_H
```

Finally, execute the following commands:

```
$ autoconf

$ ./configure

checking for gcc... gcc

...

checking for unistd.h... yes

checking for unistd.h... (cached) yes

checking foobar.h usability... no

checking foobar.h presence... no

checking for foobar.h... no

configure: creating ./config.status

config.status: creating config.h

$

$ cat config.h

/* config.h.  Generated from ...  */

#define HAVE_UNISTD_H 1

/* #undef HAVE_FOOBAR_H */
```

You can see that `config.status` generated a `config.h` file from your `config.h.in` template file. The contents of this header file are based on the checks executed by the `configure` script. Since the shell code generated by `AC_CHECK_HEADERS([unistd.h foobar.h])` was able to locate a `unistd.h` header file in the standard include directory, the corresponding `#undef` statement was converted into a `#define` statement. Of course, no `foobar.h` header was found in the system include directory, as you can also see by the output of `configure`, so it's definition was left commented out in the template.
Thus, you may add this sort of code to appropriate C source files in your project:

```
#if HAVE_CONFIG_H
```

```
# include <config.h>
#endif


#if HAVE_UNISTD_H
# include <unistd.h>
#endif
```

Using Autoheader to generate an include file template

Maintaining your `config.h.in` template is more pain than necessary. After all, most of the information you need is already encapsulated in your `configure.ac` script, and the format of `config.h.in` is very strict. For example, you may not have any leading or trailing white space on the `#undef` lines.

Fortunately, the `autoheader` utility will generate an include header template for you based on your `configure.ac` file contents. Back to the command prompt for another quick experiment. This one is easy--just *delete* your `config.h.in` template before you run `autoheader` and `autoconf`, like this:

```
$ rm config.h.in
$ autoheader
$ autoconf
$ ./configure
checking for gcc... gcc
...
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking foobar.h usability... no
checking foobar.h presence... no
checking for foobar.h... no
configure: creating ./config.status
config.status: creating config.h
$ cat config.h
```

```
/* config.h. Generated from config.h.in...  */

/* config.h.in. Generated from configure.ac... */

...

/* Define to 1 if you have... */

/* #undef HAVE_FOOBAR_H */


/* Define to 1 if you have... */

#define HAVE_UNISTD_H 1


/* Define to the address where bug... */

#define PACKAGE_BUGREPORT ""


/* Define to the full name of this package. */

#define PACKAGE_NAME "test"


/* Define to the full name and version... */

#define PACKAGE_STRING "test 1.0"


/* Define to the one symbol short name... */

#define PACKAGE_TARNAME "test"


/* Define to the version... */

#define PACKAGE_VERSION "1.0"


/* Define to 1 if you have the ANSI C... */

#define STDC_HEADERS 1
```

NOTE: Here again, I encourage you to use `autoreconf`, which will automatically run autoheader for you if it notices an expansion of the `AC_CONFIG_HEADERS` macro in your `configure.ac` script.

You may also want to take a peek at the `config.h.in` template file generated by autoheader. In the meantime, here's a much more realistic example of using a generated `config.h` file for the sake of portability of project source code.

```
AC_INIT([test], [1.0])

AC_CONFIG_HEADERS([config.h])

AC_CHECK_HEADERS([dlfcn.h])

AC_OUTPUT
```

The `config.h` file is obviously intended to be included in your source code in locations where you might wish to test a configured option in the code itself using the C preprocessor. Using this `configure.ac` script, Autoconf will generate a `config.h` header file with appropriate definitions for determining, at compile time, if the current system provides the `dlfcn` interface. To complete the portability check, you can add the following code to a source file that uses dynamic loader functionality in your project:

```
#if HAVE_CONFIG_H

# include <config.h>

#endif



#if HAVE_DLFCN_H

# include <dlfcn.h>

#else

# error Sorry, this code requires dlfcn.h.

#endif

...

#if HAVE_DLFCN_H

   handle = dlopen(

      "/usr/lib/libwhatever.so", RTLD_NOW);

#endif
```

```
...
```

If you already had code that included `dlfcn.h` then `autoscan` will have generated a `configure.ac` call to `AC_CHECK_HEADERS`, which contains `dlfcn.h` as one of the header files to be checked. Your job as the maintainer is to add the conditional to your source code around the existing use of the `dlfcn.h` header inclusion and the `libdl.so` API calls. This is the crux of Autoconf-provided portability.

Your project may be able to get along at compile time without the dynamic loader functionality if it must, but it would be nice to have it. Perhaps, your project will function in a limited manner without it. Sometimes you just have to bail out with a compiler error (as this code does) if the key functionality is missing. Often this is an acceptable first-attempt solution, until someone comes along and adds support to the code base for some other dynamic loader service that is perhaps available on non-dlfcn-oriented systems.

NOTE: If you have to bail out with an error, it's best to do so at configuration time, rather than at compile time. The general rule of thumb is to bail out as early as possible. I'll cover examples of this sort of activity shortly.

One obvious flaw in this source code is that `config.h` is only included if `HAVE_CONFIG_H` is defined in your compilation environment. But wait...doesn't that definition happen in `config.h`?! Well, no, not in the case of this particular definition. `HAVE_CONFIG_H` must be either defined by you manually, if you're writing your own makefiles, or automatically by Automake-generated makefiles on the compiler command line. (Are you beginning to get the feeling that Autoconf really shines when used in conjunction with Automake?)

`HAVE_CONFIG_H` is part of a string of definitions passed on the compiler command line in the Autoconf substitution variable `@DEFS@`. Before Autoheader and `AC_CONFIG_HEADERS`, all of the compiler configuration macros were added to the `@DEFS@` variable. You can still use this method if you don't use `AC_CONFIG_HEADERS` in `configure.ac`. but it's not the recommended method nowadays, mainly because a large number of definitions make for a very long compiler command line.

Back to `VPATH` builds for a moment

Regarding `VPATH` builds, I haven't yet covered how to get the preprocessor to properly locate my generated `config.h` file. This file, being a generated file, will be found in the same relative position in the build directory structure as its counterpart template file, `config.h.in`. The template is located in the top-level source directory (unless you choose to put it somewhere else), so the generated file will be in the top-level build directory. Well, that's easy enough--it's always one level up from the generated `src/Makefile`.

Consider where I might have include files in this project. I might add an internal header file to the current source directory. I obviously now have a `config.h` file in my top-level build directory. I might also create a top-level source include directory for library interface header files. In which order should I care about these files?

The order I place include directives (`-I<path>`) options on the compiler command line is the order which they will be searched. The proper preprocessor include paths should include the current build directory (`.`), the source directory (`$(srcdir)`), and the top-level build directory (`..`), in that order:

```
...

jupiter: main.c

        gcc -g -O0 -I. -I$(srcdir) -I..\
```

```
          -o $@ $(srcdir)/main.c

...
```

It appears that I now need an additional rule of thumb for `VPATH` builds:

- Add preprocessor commands for the current build and associated source and top-level build directories, in that order.

Checks for compilers

The `AC_PROG_CC` macro ensures that I have a working C language compiler. This call was added to `configure.scan` when `autoscan` noticed that I had C source files in my project directory. If I'd had files suffixed with "`.cxx`" or "`.C`" (an upper-case "`.C`" extension indicates a C++ source file), it would have inserted a call to the `AC_PROG_CXX` macro, as well as a call to `AC_LANG([C++])`.

This macro looks for `gcc` and then `cc` in the system search path. If neither of these are found, it looks for other C compilers. When a compatible compiler is found, it sets a well-known variable, `$CC` to the full path of the program, with options for portability, if necessary.

`AC_PROG_CC` accepts an optional parameter containing an ordered list of compiler names. For example, if you used `AC_PROG_CC([cc cl gcc])`, then the macro would expand into shell code that searched for `cc`, `cl` and `gcc`, in that order.

The `AC_PROG_CC` macro also defines the following Autoconf substitution variables:

- `@CC@` (full path of compiler)
- `@CFLAGS@` (eg., -g -O2 for gcc)
- `@CPPFLAGS@` (empty by default)
- `@EXEEXT@` (eg., .exe)
- `@OBJEXT@` (eg., .o)

`AC_PROG_CC` configures these substitution variables, but unless I used them in my `Makefile.in` templates, I'm just wasting time running `configure`. I'll add a few of these as `make` variables to my `src/Makefile.in` template, and then consume them, like this:

```
# Tool-related substitution variables

CC              = @CC@

CFLAGS          = @CFLAGS@

CPPFLAGS        = @CPPFLAGS@

...

jupiter: main.c

        $(CC) $(CFLAGS) $(CPPFLAGS)\

         -I. -I$(srcdir) -I..\

         -o $@ $(srcdir)/main.c
```

Checking for other programs

Now, let's return to the AC_PROG_INSTALL macro. As with the AC_PROG_CC macro, the other AC_PROG_* macros set and then substitute (using AC_SUBST) various environment variables that point to the located utility. To make use of this check, you need to use these Autoconf substitution variables in your Makefile.in templates, just as I did with CC, CFLAGS, and CPPFLAGS above:

```
...

# Tool-related substitution variables

CC              = @CC@

CFLAGS          = @CFLAGS@

CPPFLAGS        = @CPPFLAGS@

INSTALL         = @INSTALL@

INSTALL_DATA    = @INSTALL_DATA@

INSTALL_PROGRAM= @INSTALL_PROGRAM@

INSTALL_SCRIPT = @INSTALL_SCRIPT@

...

install:

        $(INSTALL) -d $(DESTDIR)$(bindir)/jupiter

        $(INSTALL_PROGRAM) -m 0755 jupiter \

         $(DESTDIR)$(bindir)/jupiter

...
```

The value of @INSTALL@ is obviously the path of the located install script. The value of @INSTALL_DATA@ is ${INSTALL} -m 0644. Now, you'd think that the values of @INSTALL_PROGRAM@ and @INSTALL_SCRIPT@ would be ${INSTALL} -m 0755, but they're not. These are just set to ${INSTALL}. Oversight? I don't know.

Other important utility programs you might need to check for are lex, yacc, sed, awk, etc. If so, you can add calls to AC_PROG_LEX, AC_PROG_YACC, AC_PROG_SED, or AC_PROG_AWK yourself. There are about a dozen different programs you can check for using these more specialized macros. If such a program check fails, then the resulting configure script will fail with a message indicating that the required utility could not be found, and that the build may not continue until it's been properly installed.

As with the other program and compiler checks in Makefile.in templates, you should use the make variables $(LEXX) and $(YACC) to invoke these tools (note that Automake does this for you), as these Autoconf macros will set the values of these variables according to the tools it finds installed on your system *if they are not already set in your environment*.

Now, this is a key aspect of `configure` scripts generated by Autoconf--you may *always* override anything `configure` will do to your environment by exporting or setting an appropriate output variable *before* you execute `configure`.

For example, perhaps you would like to build with a very specific version of `bison` that you've installed in your own home directory:

```
$ cd jupiter

$ YACC="$HOME/bin/bison -y" ./configure

$ ...
```

This will ensure that `YACC` is set the way you want for your makefiles, and that `AC_PROG_YACC` does essentially nothing in your `configure` script.

If you need to check for the existence of a program not covered by these more specialized macros, you can call the generic `AC_CHECK_PROG` macro, or you can write your own special purpose macro (I'll cover writing macros in Chapter 9).

Key points to take away:

- `AC_PROG_*` macros check for the existence of programs.
- If a program is found, a substitution variable is created.
- Use these variables in your `Makefile.in` templates to execute the program.

A common problem with Autoconf

Here's a common problem that developers new to the Autotools consistently encounter. Take a look at the formal definition of AC_CHECK_PROG found in the GNU Autoconf manual. NOTE: In this case, the square brackets represent optional parameters, not Autoconf quotes.:

`AC_CHECK_PROG(variable, prog-to-check-for, value-if-found, [value-if-not-found], [path], [reject])`

Check whether program `prog-to-check-for` exists in PATH. If it is found, set `variable` to `value-if-found`, otherwise to `value-if-not-found`, if given. Always pass over `reject` (an absolute file name) even if it is the first found in the search path; in that case, set `variable` using the absolute file name of the `prog-to-check-for` found that is not `reject`. If `variable` was already set, do nothing. Calls AC_SUBST for `variable`.

I can extract the following clearly defined functionality from this description:

- If `prog-to-check-for` is found in the system search path, then `variable` is set to `value-if-found`, otherwise it's set to `value-if-not-found`.
- If `reject` is specified (as a full path), then skip it if it's found first, and continue to the next matching program in the system search path.
- If `reject` is found first in the path, and then another match is found besides `reject`, set `variable` to the absolute path name of the second (non-reject) match.
- If `variable` is already set by the user in the environment, then `variable` is left untouched (thereby allowing the user to override the check by setting `variable` before running `autoconf`).
- `AC_SUBST` is called on `variable` to make it an Autoconf substitution variable.

At first read, there appears to be a terrible conflict of interest here: We can see in point 1 that `variable` will be set to one or the other of two specified values, based on whether or not `prog-to-check-for` is found in the system search path. But

then in point 3 it states that `variable` *will be set to the full path of some program, but only if* `reject` *is found first and skipped. Clearly the documentation needs a little work.*
Discovering the real functionality of `AC_CHECK_PROG` *is as easy as reading a little shell script. While you could spend your time looking at the definition of* `AC_CHECK_PROG` *in* `/usr/share/autoconf/autoconf/programs.m4`*, the problem with this approach is that you're one level removed from the actual shell code performing the check. Wouldn't it be better to just look at the resulting shell script generated by* `AC_CHECK_PROG`*? Okay, then modify your new* `configure.ac` *file in this manner:*

```
...
AC_PREREQ(2.59)
AC_INIT([jupiter], [1.0],
   [jupiter-devel@lists.example.com])
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC
AC_CHECK_PROG([bash_var], [bash], [yes],
   [no],, [/usr/sbin/bash])
...
```

Now just execute `autoconf` *and then open the resulting* `configure` *script and search for something specific to the definition of* `AC_CHECK_PROG`*. I used the string "ac_cv_prog_bash_var", a shell variable generated by the macro call. You may have to glance at the definition of a macro to find reasonable search text:*

```
$ autoconf
$ vi -c /ac_cv_prog_bash_var configure
...
# Extract the first word of "bash", so it can be
#   a program name with args.
```

```
set dummy bash; ac_word=$2

echo "$as_me:$LINENO: checking for $ac_word" >&5

echo $ECHO_N "checking for $ac_word... $ECHO_C"\

 >&6

if test "${ac_cv_prog_bash_var+set}" = set; then

  echo $ECHO_N "(cached) $ECHO_C" >&6

else

  if test -n "$bash_var"; then

  # Let the user override the test.

  ac_cv_prog_bash_var="$bash_var"

else

  ac_prog_rejected=no

as_save_IFS=$IFS; IFS=$PATH_SEPARATOR

for as_dir in $PATH

do

  IFS=$as_save_IFS

  test -z "$as_dir" && as_dir=.

  for ac_exec_ext in ''\

 $ac_executable_extensions;

  do

  if $as_executable_p\

 "$as_dir/$ac_word$ac_exec_ext"; then

    if test "$as_dir/$ac_word$ac_exec_ext" =\

 "/usr/sbin/bash"; then

       ac_prog_rejected=yes

       continue
```

```
      fi
     ac_cv_prog_bash_var="yes"
     echo "$as_me:$LINENO: found\
  $as_dir/$ac_word$ac_exec_ext" >&5
     break 2
   fi
done
done

if test $ac_prog_rejected = yes; then
  # We found a bogon in the path, so make sure
  # we never use it.
  set dummy $ac_cv_prog_bash_var
  shift
  if test $# != 0; then
    # We chose a different compiler from the
    # bogus one. However, it has the same
    # basename, so the bogon will be chosen
    # first if we set bash_var to just the
    # basename; use the full file name.
    shift
    ac_cv_prog_bash_var=\
 "$as_dir/$ac_word${1+' '}$@"
  fi
fi
  test -z "$ac_cv_prog_bash_var" &&\
```

```
  ac_cv_prog_bash_var="no"

fi

fi

bash_var=$ac_cv_prog_bash_var

if test -n "$bash_var"; then

  echo "$as_me:$LINENO: result: $bash_var" >&5

echo "${ECHO_T}$bash_var" >&6

else

  echo "$as_me:$LINENO: result: no" >&5

echo "${ECHO_T}no" >&6

fi

...
```

Wow! You can immediately see by the opening comment that `AC_CHECK_PROG` *has some undocumented functionality: You can pass in arguments with the program name if you wish. But why would you want to? Well, look farther. You can probably fairly accurately deduce that the* `reject` *parameter was added into the mix in order to allow your* `configure` *script to search for a particular version of a tool. (Could it possibly be that someone might really rather use the GNU C compiler instead of the Solaris C compiler?)*
In fact, it appears that `variable` *really is set based on a tri-state condition. If* `reject` *is not used, then* `variable` *can only be either* `value-if-found` *or* `value-if-not-found`*. But if* `reject` *is used, then* `variable` *can also be the full path of the first program found that is not reject! Well, that is exactly what the documentation stated, but examining the generated code yields insight into the authors' intended use of this macro. We probably should have called* `AC_CHECK_PROG` *this way, instead:*

```
AC_CHECK_PROG([bash_shell],[bash -x],[bash -x],,,

              [/usr/sbin/bash])
```

Now it makes more sense, and you can see by this example that the manual is in fact accurate, if not clear. If `reject` *is not specified, and bash is found in the system path, then* `bash_shell` *will be set to* `bash -x`*. If it's not found in the system path, then* `bash_shell` *will be set to the empty string. If, on the other hand,* `reject` *is specified, and the undesired version of bash is found first in the path, then* `bash_shell` *will be set to the full path of the next version found in the path, along with the originally specified arguments (-x). The* `bash_shell` *variable may now be* *used by the rest of our script to run the desired bash shell, if it doesn't test out as empty. Wow! No wonder it was hard to document in a way that's easy to understand! But quite frankly, a good example of the intended use of this macro, along with a couple of sentences of explanation would have made all the difference.*

Checks for libraries and header files

Does your project rely on external libraries? Most non-trivial projects do. If you're lucky, your project relies only on libraries that are already widely available and ported to most platforms.

The choice to use an external library or not is a tough one. On the one hand, you'll want to reuse code that provides functionality--perhaps significant functionality that you need and don't really have the time or expertise to write yourself. Reuse is one of the hallmarks of the free software world.

On the other hand, you don't want to depend on functionality that may not exist on all of the platforms you wish to target, or that requires significant porting effort on your part to make these libraries available on all of your target platforms.

Occasionally, library-based functionality can exist in slightly different forms on different platforms. These different forms may be functionally compatible, but have different API signatures. For example, POSIX threads (pthreads) versus a native threading library. For basic multi-threading functionality, many threading libraries are similar enough to be almost drop-in replacements of each other.

To illustrate this concept, I'll add some trival multi-threading capabilities to the Jupiter project. I want to have `jupiter` print its message using a background thread. To do this, I'm going to need to add the pthreads library to my project build system. If I weren't using the Autotools, I'd just add it to my linker command line in the makefile:

```
jupiter: main.c

        $(CC) ... -lpthreads ...
```

But what if a system doesn't support pthreads? I might want to support native threads on a non-pthreads system--say Solaris native threads, using the `libthreads` library.
To do this, I'll first modify my `main.c` file such that the printing happens in a secondary thread, like this:

`src/main.c`

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>



static void * print_it(void * data)
```

```
{
   printf("Hello from %s!\n", (char *)data);
   return 0;
}


int main(int argc, char * argv[])
{
   pthread_t tid;
   pthread_create(&tid, 0, print_it, argv[0]);
   pthread_join(tid, 0);
   return 0;
}
```

Now, this is clearly a ridiculous use of a thread. Nonetheless, it *is* the prototypical form of thread usage. Consider the case where `print_it` did some long calculation, and `main` had other things to do while `print_it` performed this calculation. On a multi-processor machine, this could literally double the throughput of such a program.

What we now need is a way of determining which libraries should be added to the compiler command line. Enter Autoconf and the `AC_CHECK_*` macros. The `AC_SEARCH_LIBS` macro allows us to check for key functionality within a list of libraries. If the function exists within one of the specified libraries, then an appropriate command line option is added to the `@LIBS@` substitution variable. The `@LIBS@` variable should be used in a `Makefile.in` template on the compiler (linker) command line. Here is the formal definition of `AC_SEARCH_LIBS`, again from the manual:

`AC_SEARCH_LIBS(function, search-libs, [action-if-found], [action-if-not-found], [other-libraries])` Search for a library defining `function` if it's not already available. This equates to calling `AC_LINK_IFELSE([AC_LANG_CALL([], [function])])` first with no libraries, then for each library listed in `search-libs`. Add `-llibrary` to LIBS for the first library found to contain function, and run `action-if-found`. If `function` is not found, run `action-if-not-found`. If linking with the library results in unresolved symbols that would be resolved by linking with additional libraries, give those libraries as the `other-libraries` argument, separated by spaces: e.g., `-lXt -lX11`. Otherwise, this macro fails to detect that `function` is present, because linking the test program always fails with unresolved symbols.

Wow, that's a lot of stuff for one macro. Are you beginning to see why the generated `configure` script is so large? Essentially, what you get by calling `AC_SEARCH_LIBS` for a particular function is that the proper linker command line arguments (eg., `-lpthread`), for linking with a library containing the desired function, are added to a substitution variable called `@LIBS@`. Here's how I'll use `AC_SEARCH_LIBS` in my `configure.ac` file:

`configure.ac`

```
...

# Checks for libraries.

AC_SEARCH_LIBS([pthread_create], [pthread])

...
```

Of course, I'll have to modify `src/Makefile.in` again to make proper use of the now populated `LIBS` variable:

```
...

# Tool-related substitution variables

CC              = @CC@

LIBS            = @LIBS@

CFLAGS          = @CFLAGS@

CPPFLAGS        = @CPPFLAGS@

...

jupiter: main.c

        $(CC) $(CFLAGS) $(CPPFLAGS)\

         -I. -I$(srcdir) -I..\

         -o $@ $(srcdir)/main.c $(LIBS)

...
```

Note that I added `$(LIBS)` *after* the source file on the compiler command line. Generally, the linker cares about object file order, and searches them for required functions in the order they are specified on the command line. Since I want `main.c` to be the primary source of object code for `jupiter`, I'll continue to add additional objects, including libraries, after this file on the command line.

Right or just good enough?

I could just stop at this point. I've done enough to make this build system properly use pthreads on most systems. If a library is needed, it'll be added to the `@LIBS@` variable, and subsequently used on my compiler command line. In fact, this is the point at which many maintainers *would* stop. The problem is that stopping here is just about the build-system equivalent of not checking the return value of `malloc` in a C program (and there are many developers out there who don't give this process the credit it deserves either). It *usually* works fine. It's just during those few cases where it fails that you have a real problem.

Well, I want to provide a good user experience, so I'll take Jupiter's build system to the "next level". However, in order to do this, I need to make a design decision: In case `configure` fails to locate a pthread library on a user's system, should I fail the build process, or build a `jupiter` program without multi-threading? If I fail the build, it will generally be obvious to the user, because the build has stopped with an error message--although, perhaps not a very user-friendly one. At this point, either the compile process or the link process will fail with a cryptic error message about a missing header file or an undefined symbol. If I choose to build a single-threaded version of `jupiter`, I should probably display some clear message that I'm moving forward without threads, and why.
There's another potential problem also. Some users' systems may have a `pthread` library installed, but not have the `pthread.h` header file installed properly. This can happen for a variety of reasons, but the most common is that the executable package was installed, but not the developer package. Executable binaries are often packaged independently of static libraries and header files. Executables are installed as part of a dependency chain for a higher level consuming application, while developer packages are often only installed directly by a user. For this reason, Autoconf provides checks for both libraries and header files. The `AC_CHECK_HEADERS` macro is used to ensure the existence of a particular header file.
Autoconf checks are very thorough. They generally not only ensure the existence of a file, but also that the file is in fact the one you're looking for. They do this by allowing you to make some assertions about the file, which are then verified by the macro. Additionally, the `AC_CHECK_HEADERS` macro doesn't just scan the file system for the requested header. It actually builds a short test program in the appropriate language, and then compiles it to ensure that the compiler can both find the file, and use it. Similarly, `AC_SEARCH_LIBS` is built around an attempt to link to the specified libraries, and import the requested symbols.
Here is the formal definition of `AC_CHECK_HEADERS`, as found in the GNU Autoconf manual:
`AC_CHECK_HEADERS(header-file..., [action-if-found], [action-if-not-found], [includes = 'default-includes'])` For each given system header file `header-file` in the blank-separated argument list that exists, define `HAVE_header-file` (in all capitals). If `action-if-found` is given, it is additional shell code to execute when one of the header files is found. You can give it a value of `break` to break out of the loop on the first match. If `action-if-not-found` is given, it is executed when one of the header files is not found.
Normally, this macro is called only with a list of desired header files in the first argument. Remaining arguments are optional and are not often used. The reason for this is that the macro is very functional when used in this manner. I'll add a check for the pthread library using `AC_CHECK_HEADERS` to my `configure.ac` file.
If you're the jump-right-in type, then you've noticed by now that `configure.ac` already calls `AC_CHECK_HEADERS` for `stdlib.h`. No problem--I'll just add `pthread.h` to the list, using a space to separate the file names, like this:

```
...

# Checks for header files.

AC_HEADER_STDC

AC_CHECK_HEADERS([stdlib.h pthread.h])

...
```

I like to make my packages available to as many people as possible, so I'll go ahead and use the dual-mode build approach, where I can at least provide *some* form of `jupiter` program to users without pthreads. To accomplish this, I'll need to add some conditional compilation preprocessor code to my `src/main.c` file:

`src/main.c`

```
#include <stdio.h>

#include <stdlib.h>



#if HAVE_PTHREAD_H

# include <pthread.h>

#endif



static void * print_it(void * data)

{

   printf("Hello from %s!\n", (char *)data);

   return 0;

}



int main(int argc, char * argv[])

{

#if HAVE_PTHREAD_H

   pthread_t tid;

   pthread_create(&tid, 0, print_it, argv[0]);

   pthread_join(tid, 0);

#else

   print_it(argv[0]);

#endif

   return 0;
```

```
}
```

In this version of `main.c`, I've added a couple of conditional checks for the existence of the header file. The `HAVE_PTHREAD_H` macro will be defined to the value `1` in the `config.h.in` template, if the `AC_CHECK_HEADERS` macro locates the `pthread.h` header file, otherwise the definition will be added as a comment in the template. Thus, I'll need to include the `config.h` file at the top of my `main.c` file:

```
#if HAVE_CONFIG_H

# include <config.h>

#endif

...
```

Recall that `HAVE_CONFIG_H` must be defined on the compiler command line, and that Autoconf populates the `@DEFS@`, substitution variable with this definition, if `config.h` is available. If you choose not to use the `AC_CONFIG_HEADERS` macro in your `configure.ac`, then `@DEFS@` will contain all of the definitions generated by all of the various check macros you do use. In this example, I've used `AC_CONFIG_HEADERS`, so my `config.h.in` template will contain most of these definitions, and `@DEFS@` will only contain `HAVE_CONFIG_H`. Again, this is a nice way to go because it significantly shortens the compiler command line. An additional benefit is that it becomes very simple to take a snapshot of the template, and modify it by hand for non-Autotools platforms, such as Microsoft Windows, which doesn't require as dynamic of a configuration process as does Unix/Linux. I'll go ahead and make the required changes to my `src/Makefile.in` template, like this:

`src/Makefile.in`

```
...

# Tool-related substitution variables

CC              = @CC@

DEFS            = @DEFS@

LIBS            = @LIBS@

CFLAGS          = @CFLAGS@

CPPFLAGS        = @CPPFLAGS@

...

jupiter: main.c

        $(CC) $(CFLAGS) $(DEFS) $(CPPFLAGS)\
```

```
         -I. -I$(srcdir) -I..\

         -o $@ $(srcdir)/main.c $(LIBS)

...
```

Now, I have everything I need to conditionally build the `jupiter` program. If the end-user's system has pthread functionality, she'll get a version of `jupiter` that uses multiple threads of execution, otherwise, she'll have to settle for serialized execution. The only thing left is to add some code to the `configure.ac` script that displays a message during configuration, indicating that it's defaulting to serialized execution if the library is not found.

Another point to consider here is what it means to have the header file installed, but no library. This is very unlikely, but it can happen. However, this is easily remedied by simply skipping the header file check entirely if the library isn't found. We'll reorganize things a bit to handle this case also:

`configure.ac`

```
...

# Checks for libraries.

have_pthreads=no

AC_SEARCH_LIBS([pthread_create], [pthread],

  [have_pthreads=yes])



# Checks for header files.

AC_HEADER_STDC

AC_CHECK_HEADERS([stdlib.h])



if test "x${have_pthreads}" = xyes; then

  AC_CHECK_HEADERS([pthread.h], [],

    [have_pthreads=no])

fi



if test "x${have_pthreads}" = xno; then
```

```
    echo "------------------------------------------"
    echo " Unable to find pthreads on this system.  "
    echo " Building a single-threaded version.      "
    echo "------------------------------------------"
  fi
  ...
```

I'll run `autoreconf` and `configure` and see what additional output I get now:

```
$ autoreconf
$ ./configure
checking for gcc... gcc
...
checking for library... pthread_create... -lpthread
...
checking pthread.h usability... yes
checking pthread.h presence... yes
checking for pthread.h... yes
configure: creating ./config.status
config.status: creating Makefile
...
```

Of course, if your system doesn't have pthreads, you'll get something a little different. To emulate this, I'll rename my pthreads libraries (both shared and static), and then rerun `configure`:

```
$ su
Password:
# mv /usr/lib/libpthread.so ...
# mv /usr/lib/libpthread.a ...
```

```
# exit

exit

$ ./configure

checking for gcc... gcc

...

checking for library... pthread_create... no

...

checking for stdint.h... yes

checking for unistd.h... yes

checking for stdlib.h... (cached) yes

-----------------------------------------

 Unable to find pthreads on this system.

   Building a single-threaded version.

-----------------------------------------

configure: creating ./config.status

config.status: creating Makefile

config.status: creating src/Makefile

config.status: creating config.h
```

Of course, if I had chosen to fail the build if I couldn't find the `pthread.h` header file or the pthreads libraries, then my source code would have been simpler--no need for conditional compilation. I could change my `configure.ac` file to look like this, instead:

`configure.ac`

```
...

# Checks for libraries.

have_pthreads=no

AC_SEARCH_LIBS([pthread_create], [pthread],

  [have_pthreads=yes])
```

```
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])

if test "x${have_pthreads}" = xyes; then
  AC_CHECK_HEADERS([pthread.h], [],
    [have_pthreads=no])
fi

if test "x${have_pthreads}" = xno; then
  echo "------------------------------------------"
  echo " The pthread library and header file is   "
  echo " required to build jupiter. Stopping...   "
  echo " Check 'config.log' for more information. "
  echo "------------------------------------------"
  (exit 1); exit 1;
fi
...
```

I could have used a couple of macros provided by Autoconf for the purpose of printing messages to the console: `AC_MSG_WARNING` and `AC_MSG_ERROR`, but I don't really care for these macros, because they tend to be single-line-oriented. This is especially a problem in the case of the warning message, which merely indicates that it's continuing, but it's building a single-threaded version of `jupiter`. Such a single-line message could zip right by in a large configuration process, without even being noticed by the user.

In the case where I decide to terminate with an error, this is less of a problem, because--well, I terminated. But, for the sake of consistency, I like all of my messages to look the same. There is a note in the GNU Autoconf manual indicating that some shells are not able to properly pass the value of the `exit` parameter to the parent shell, and that `AC_MSG_ERROR` has a work-around for this problem. Well, the funny code after the echo statements in this last example *is* this very work-around, copied right out of a test `configure` script that I created using `AC_MSG_ERROR`.

This last topic brings to light a general lesson regarding Autoconf checks. Checks do just that--they check. It's up to the maintainer to add code to do something based on the results of the check. This isn't strictly true, as `AC_SEARCH_LIBS` adds a library to the `@LIBS@` variable, and `AC_CHECK_HEADERS` adds a preprocessor definition to the `config.h.in` template. However, regarding the flow of control within the configure process, all such decisions are left to the developer. Keep this in mind while you're designing your `configure.ac` script, and life will be simpler for you.

Supporting optional features and packages

Alright, I've covered the cases in Jupiter where a pthreads library exists, and where it doesn't exist. I'm satisfied, at this point, that I've done just about all I can to manage both of these cases very well. But what about the case where the user wants to deliberately build a single-threaded version of `jupiter`, even in the face of an existing pthreads library? Do I add a note to Jupiter's `README` file, indicating that the user should rename her pthreads libraries in this case? I don't think so.

Autoconf provides for both optional features, and optional sub-packages with two new macros: `AC_ARG_ENABLE` and `AC_ARG_WITH`. These macros are designed to do two things: First, to add help text to the output generated when you enter "`configure --help`", and second, to check for the specified options, "`--enable-feature[=yes|no]`", and "`--with-package[=arg]`" on the configure script's command line, and then set appropriate environment variables within the script. The values of these variables may be used later in the script to set or clear various preprocessor definitions or substitution variables.

`AC_ARG_WITH` is used to control the use of optional sub-packages which may be consumed by your package. `AC_ARG_ENABLE` is used to control the inclusion or exclusion of optional features in your package. The choice to use one or the other is often a matter of perspective and sometimes simply a matter of preference, as they provide somewhat overlapping sets of functionality. For instance, in the Jupiter package, it could be justifiably argued that Jupiter's use of pthreads constitutes the use of an external package. However, it could just as well be said that asynchronous processing is a feature that might be enabled.

In fact, both of these statements are true, and which type of option you use should be dictated by a high-level architectural perspective on the software in question. For example, the pthreads library supplies more than just thread creation functions. It also provides mutexes and condition variables, both of which may be used by a library package that doesn't create threads. If a project provides a library that needs to act in a thread-safe manner within a multi-threaded process, then it will probably use one or more mutex objects. But it may never create a thread. Thus, a user may choose to disable asynchronous execution within this library package at configuration time, but the package may still need to link the pthread library in order to access the mutex functionality from an unrelated portion of the code.

From this perspective, it makes more sense to specify "`--enable-async-exec`" than "`--with-pthreads`". Indeed, from a purist's perspective, this rationale is always sound, even in cases where a project only uses pthreads to create threads. When writing software, you won't often go wrong by siding with the purist. While some of their choices may seem arbitrary--even rediculous, they're almost always vindicated at some point in the future.

So, when do you use `AC_ARG_WITH`? Generally, when a choice should be made between implementing functionality one way or another. That is, when there is a choice to use one package or another, or to use an external package, or an internal implementation. For instance, if `jupiter` had some reason to encrypt a file, it might be written to use either an internal encryption algorithm, or an external package, such as openssl. When it comes to encryption, the use of a widely understood package can be a great boon toward gaining community adoption of your package. However, it can also be a hindrance to those who don't have access to a required external package. Giving your

users a choice can make all the difference between them having a good or bad experience with your package.
These two macros have very similar signatures, so I'll just list them here together:

```
AC_ARG_WITH(package, help-string, [action-if-given], [action-if-not-given])
AC_ARG_ENABLE(feature, help-string, [action-if-given], [action-if-not-given])
```

As with many Autoconf macros, these may be used in a very simple form, where the check merely sets environment variables:

- `${withval}` and `${with_package}`
- `${enableval}` and `${enable_feature}`

They can also be used in a more complex form, where these environment variables are used by shell script in the optional arguments. In either case, as usual, the resulting variable must be used in order to act on the results of the check, or performing the check is pointless.

Coding up the feature option

Okay, I've now decided that I should use `AC_ARG_ENABLE`. Do I enable or disable the "async-exec" feature by default? The difference in how these two cases are encoded is limited to the help text and to the shell script that I put into the `action-if-not-given` argument. The help text describes the available options and the default value, and the shell script indicates what I want to have happen if the option is NOT specified. Of course, if it is specified, I don't need to assume anything.
Say I decide that asynchronous execution is a risky feature. In this case, I want to disable it by default, so I might add code like this to my `configure.ac` script:

`configure.ac`

```
...
AC_ARG_ENABLE([async-exec],
  [  --enable-async-exec     enable async exec],
  [async_exec=${enable_val}],
  [async_exec=yes])
...
```

On the other hand, if I decide that asynchronous execution is a fairly fundamental part of Jupiter, then I'd like it to be enabled by default. In this case I'd use code like this:

`configure.ac`

```
...
AC_ARG_ENABLE([async-exec],
```

```
    [  --disable-async-exec    disable async exec],

    [async_exec=${enable_val}],

    [async_exec=no])

...
```

There are a couple of really neat features of this macro that I'd like to point out:

- Regardless of the help text, the user may always use the syntactical standard formats, "`--enable-option[=yes|no]`" or "`--disable-option[=yes|no]`". In either case, the "`[=yes|no]`" portion is optional.
- Inverse logic is handled transparently--that is, the value of `${enableval}` always represents the user's answer to the question, "Should it be enabled?". For instance, even if the user enters something like "`--disable-option=no`", the value of `${enableval}` will still be set to `yes`.

These features of `AC_ARG_ENABLE` and `AC_ARG_WITH` make a maintainer's life a *lot* simpler.

Now, the only remaining question is, do I check for the library and header file regardless of the user's desire for this feature, or do I only check for them if the user indicates that the "async-exec" feature should be enabled. Well, in this case, it's purely a matter of preference, as I'm using the pthreads library only for this feature. Again, if I were also using the pthreads library for non-feature-specific reasons, then this question would be answered for me--I'd have to check for it.

In cases where I need the library even if the feature is disabled, I add the `AC_ARG_ENABLE` macro, as in the example above, and then an additional `AC_DEFINE` macro to define a `config.h` definition specifically for this feature. Since I don't really want to enable the feature if the library or header file is missing--even if the user specifically requested it--I also need to add some shell code to turn the feature off if either of these are missing:

`configure.ac`

```
...

# Checks for headers.

AC_HEADER_STDC



# Checks for command line options

AC_ARG_ENABLE([async-exec],

    [  --disable-async-exec    disable async exec],

    [async_exec=${enableval}],

    [async_exec=yes])
```

```
have_pthreads=no
AC_SEARCH_LIBS([pthread_create], [pthread],
  [have_pthreads=yes])

if test "x${have_pthreads}" = xyes; then
  AC_CHECK_HEADERS([pthread.h], [],
    [have_pthreads=no])
fi

if test "x${have_pthreads}" = xno; then
  if test "x${async_exec}" = xyes; then
    echo "---------------------------------------"
    echo "Unable to find pthreads on this system."
    echo "Building a single-threaded version.    "
    echo "---------------------------------------"
  fi
  async_exec=no
fi

if test "x${async_exec}" = xyes; then
  AC_DEFINE([ASYNC_EXEC], 1, [async exec enabled])
fi

# Checks for headers.
AC_CHECK_HEADERS([stdlib.h])
```

```
...
```

I've also added an additional test for a "`yes`" value in `async_exec` around the `echo` statements within the last test for `have_pthreads`. The reason for this is that this text really belongs to the feature, not the pthreads library test. Remember, I'm trying to create a logical separation between testing for pthreads, and testing for the requirements of the feature.

Of course, now I also have to modify `src/main.c` such that it uses this new definition, as follows:

`src/main.c`

```
...

#if HAVE_PTHREAD_H

# include <pthread.h>

#endif



static void * print_it(void * data)

{

   printf("Hello from %s!\n", (char *)data);

   return 0;

}



int main(int argc, char * argv[])

{

#if ASYNC_EXEC

   pthread_t tid;

   pthread_create(&tid, 0, print_it, argv[0]);

   pthread_join(tid, 0);

#else

   print_it(argv[0]);

#endif
```

```
    return 0;
}
```

Notice that I left the `HAVE_PTHREAD_H` check around the inclusion of the header file. This is so as to facilitate the use of `pthread.h` in other ways besides for this feature. In order to check for the library and header file only if the feature is enabled, I merely have to wrap the original check code in a test of `async_exec`, like this:

`configure.ac`

```
...
if test "x${async_exec}" = xyes; then
  have_pthreads=no
  AC_SEARCH_LIBS([pthread_create], [pthread],
    [have_pthreads=yes])

  if test "x${have_pthreads}" = xyes; then
    AC_CHECK_HEADERS([pthread.h], [],
      [have_pthreads=no])
  fi

  if test "x${have_pthreads}" = xno; then
    echo "----------------------------------------"
    echo "Unable to find pthreads on this system."
    echo "Building a single-threaded version.    "
    echo "----------------------------------------"
    async_exec=no
  fi
fi
...
```

This time, I've removed the test for `async_exec` from the `echo` statements, or more appropriately, I've moved the original check from around the `echo` statements, to around the entire set of checks.

Checks for typedefs and structures

I've spent a fair amount of time during my career writing cross-platform networking software. One key aspect of networking software is that the data sent in network packets from one machine to another needs to be formatted in an architecture-independent manner. If you're trying to use C-language structures to format network messages, one of the first road blocks you generally come to is the complete lack of basic C-language types that have the same size from one platform to another. The C language was purposely designed such that the sizes of its basic integer types are implementation-defined. The designers did this to allow an implementation to use sizes for char, short, int and long that are optimal for the platform. Well, this is great for optimizing software for one platform, but it entirely discounts the need for sized types when moving data *between* platforms.

In an attempt to remedy this shortcoming in the language, the C99 standard provides just such sized types, in the form of the `intX_t` and uintX_t types, where `X` may be one of `8`, `16`, `32` or `64`. While many compilers provide these types today, some are still lagging behind. GNU C, of course, has been at the fore front for some time now, providing the C99 sized types along with the `stdint.h` header file in which these types are supposed to be defined. As time goes by, more and more compilers will support C99 types completely. But for now, it's still rather painful to write portable code that uses these and other more recently defined integer-based types.

To alleviate the pain somewhat, Autoconf provides macros for determining whether such integer-based types exist on a user's platform, defining them appropriately if they don't exist. To ensure, for example, that uint16_t exists on your target platforms, you may use the following macro expansion in your `configure.ac` file:

```
AC_TYPE_UINT16_T
```

This macro will ensure that either `uint16_t` is defined in the appropriate header files (`stdint.h`, or `inttypes.h`), or that `uint16_t` is defined in `config.h` to an appropriate basic integer type that actually *is* 16 bits in size and unsigned in nature.

The compiler tests for such integer-based types is done almost universally by a generated `configure` script using a bit of C code that looks like this:

```
...

int main()

{

   static int test_array

      [1 - 2 * !((uint16_t) -1 >> (16 - 1) == 1)];

   test_array[0] = 1;

   return 0;
```

```
}
```

Now, if you study this code carefully, you'll notice that the important line is the one on which `test_array` is declared (*Note that I've wrapped this line for publication format purposes*). Autoconf is relying on the fact that all C compilers will generate an error if you attempt to define an array with a negative size. An even more thorough examination of the bracketed expression will prove to you that this expression really is a compile-time expression. I don't know if this could have been done with simpler syntax or not, but it's a fact proven over the last several years, that this code does the trick on all compilers currently supported by Autoconf--which is most of them. The array is defined with a non-negative size if (and only if) the following two conditions are met:

- `uint16_t` is in fact defined in one of the included header files.
- the actual size of `uint16_t` really is 16 bits; no more, no less.

Code that relies on the use of this macro might contain the following construct:

```
#if HAVE_CONFIG_H

# include <config.h>

#endif

#if HAVE_STDINT_H

# include <stdint.h>

#endif

...

#if defined UINT16_MAX || defined uint16_t

// code using uint16_t

#else

// complicated alternative using >16-bit unsigned

#endif
```

There are a few dozen such type-checks available in Autoconf. You should familiarize yourself with Section 5.9 of the GNU Autoconf manual, so that you have a working knowledge of what's available. I recommend you don't commit such checks to memory, but rather just know about them, so that they'll come to mind when you need to use them. Then go look them up for the exact syntax, when you do need them.

In addition to these type-specific checks, there is also a generic type check macro, `AC_CHECK_TYPES`, which allows you to specify a comma-separated list of questionable types that your project needs. Note that this list is comma-separated, not space separated, as in the case of most of these sorts of check lists. This is because type definitions (like `struct fooble`) may have embedded spaces. Since they are commadelimited, you will need to *always* use the square bracket quotes around this parameter--that is, if you list more than one type in the parameter.

`AC_CHECK_TYPES(types, [action-if-found], [action-if-not-found], [includes = 'default-includes'])`

If you don't specify a list of include files in the last parameter, then the default includes are used in the compiler test. The default includes are used via the macro `AC_INCLUDES_DEFAULT`, which is defined as follows (in version 2.62 of Autoconf):

```
#include <stdio.h>

#ifdef HAVE_SYS_TYPES_H

# include <sys/types.h>

#endif

#ifdef HAVE_SYS_STAT_H

# include <sys/stat.h>

#endif

#ifdef STDC_HEADERS

# include <stdlib.h>

# include <stddef.h>

#else

# ifdef HAVE_STDLIB_H

# include <stdlib.h>

# endif

#endif

#ifdef HAVE_STRING_H

# if !defined STDC_HEADERS && defined HAVE_MEMORY_H

# include <memory.h>

# endif

# include <string.h>

#endif

#ifdef HAVE_STRINGS_H
```

```
# include <strings.h>

#endif

#ifdef HAVE_INTTYPES_H

# include <inttypes.h>

#endif

#ifdef HAVE_STDINT_H

# include <stdint.h>

#endif

#ifdef HAVE_UNISTD_H

# include <unistd.h>

#endif
```

If you know that your type is not defined in one of these header files, then you should specify one or more include files to be included in the test, like this:

```


AC_CHECK_TYPES([struct doodah], [], [], [

#include<doodah.h>

#include<doodahday.h>])
```

The interesting thing to note here is the way I wrapped the last parameter of the macro over three lines in `configure.ac`, with no indentation. This time I didn't do it for publication reasons. This text is included verbatim in the test source file. Since some compilers have a problem with placing the POUND SIGN (`#`) anywhere but the first column, it's a good idea to tell Autoconf to start each include line in column one, in this manner.

Admittedly, these are the sorts of things that developers complain about regarding Autoconf. When you do have problems with such syntax, your best friend is the `config.log` file, which contains the exact source code for all failed tests. You can simply look a this log file to see how Autoconf formatted the test, possibly incorrectly, and then fix your check in `configure.ac` accordingly.

The `AC_OUTPUT` macro

The `AC_OUTPUT` macro expands into the shell code that generates the `configure` script, based on all the data specified in all previous macro expansions. The important thing to note here is that all other macros must be used before `AC_OUTPUT` is expanded, or they will be of little value to your `configure` script.

Additional shell script may be placed in `configure.ac` after `AC_OUTPUT` is expanded, but this additional code will not affect the configuration or the file generation performed by `config.status`.

I like to add some echo statements after `AC_OUTPUT` to indicate to the user how the system is configured, based on their specified command line options, and perhaps additional useful targets for `make`. For example, one of my projects has the following text after `AC_OUTPUT` in `configure.ac`:

```
...

echo \

"-------------------------------------------------



 ${PACKAGE_NAME} Version ${PACKAGE_VERSION}



 Prefix: '${prefix}'.

 Compiler: '${CC} ${CFLAGS} ${CPPFLAGS}'



 Package features:

   Async Execution: ${async_exec}



 Now type 'make @<:@<target>@:>@'

   where the optional <target> is:

     all                - build all binaries

     install            - install everything



--------------------------------------------------"
```

This is a really handy `configure` script feature, as it tells the user at a glance just what happened during configuration. Since variables such as `debug` are set on or off based on configuration, the user can see if the configuration he asked for actually took place.
By the way, in case you're wondering what those funny character sequences are around the word <target>, they're called quadrigraph sequences or simply quadrigraphs, and serve the same purpose as escape sequences. Quadrigraphs are a little more reliable than escaped characters or escape sequences because they're never subject to ambiguity. They're converted to proper characters at a very late stage by M4, and so are not subject to mis-interpretation.

The sequence, `@<:@` is the quadrigraph sequence for the open square bracket ([) character, while `@:>@` is the quadrigraph for the close square bracket (]) character. These quadrigraphs will *always* be output by Autoconf (M4) as literal bracket characters. This keeps Autoconf from interpreting them as Autoconf quote characters.
There are a few other quadrigraphs. I'll show you some of them in Chapter 9 when I begin to discuss the process of writing your own Autoconf macros. If you're interested, check out section 8.1.5 of the GNU Autoconf manual.

NOTE: Version 2.62 of Autoconf does a much better job of deciphering the user's intent with respect to the use of square brackets than previous versions of Autoconf. Where you might have needed to use a quadrigraph in the past to force Autoconf to display a square bracket, you may now use the character itself. Most of the problems the occur are a result of not properly quoting arguments.

Does (project) size matter?

An issue that might have occurred to you by now is the size of my toy project. I mean, c'mon! One source file?! But, I've used `autoscan` to autoconfiscate projects with several hundred C++ source files, and some pretty complex build steps. It takes a few seconds longer to run `autoscan` on a project of this size, but it works just as well. For a basic build, the generated `configure` script only needed to be touched up a bit--project name, version, etc.
To add in compiler optimization options for multiple target tool sets, it took a bit more work. I'll cover these sorts of issues in Chapter 6 where I'll show you how to autoconfiscate a real project.

Summary

In this chapter, I've covered about a tenth of the information in the GNU Autoconf manual, but in much greater analytical detail than the manual. For the developer hoping to quickly bootstrap into Autoconf, I believe I've covered one of the more important "tenths". But this statement in no way alleviates a responsible software engineer from studying the other nine tenths--as time permits, of course.

For example, I didn't go into detail about the differences between searching for a function and searching for a library. In general, `AC_SEARCH_LIBS` should be used to check for a function you need, but expect in one or more libraries. The `AC_FUNC_*` macros are available to check for very specfic portability-related functionality, such as `AC_FUNC_ALLOCA`, which exists on some platforms, but not others. The `AC_CHECK_FUNC` macro should be used, if a particular function is not supported by one of the more specific `AC_FUNC_*` macros. I recommend reading through Section 5.5 of the GNU Autoconf manual to familiarize yourself with what's available within these special function checks.
Another topic on which I didn't spend much time was that of checking for compiler charactaristics. Section 5.10 of the GNU Autoconf manual covers these issues completely. Given what you've learned after reading this chapter, reading these sections of the manual should be pretty straight-forward.

In fact, once you're comfortable with the material in this and the preceding chapters of this book, I'd highly recommend spending a fair amount of time in Chapter 5 of the GNU Autoconf manual. Doing so will make you the Autoconf expert you never thought you could be, by filling in all of the missing details.

The next chapter takes us aways from Autoconf for a while, as we get into Automake, an Autotools tool chain add-on enhancement for the make utility.

Source archive

Download the attached source archive for the original sources associated with this chapter.

‹ Chapter 2: Project management and the GNU coding standards up Chapter 4: Automatically writing makefiles with Automake ›

Chapter 4: Automatically writing makefiles with Automake

Wed, 2008-06-25 01:50 -- John Calcote

Most of the general complaints I've ever seen aimed at the Autotools are ultimately associated with Automake, in the final analysis. The reason for this is simple: Automake provides the highest level of abstraction over the build system. This high level of abstraction is both apparent, and actual. And yet a solid understanding of the inner workings of Automake can provide you with the one of the most satisfying auto-generated build system experiences, because you can feel comfortable using the features of Automake to their maximum potential, and extending it where your projects require.

This chapter has downloads!
Shortly after Autoconf was well on its way to success in the GNU world, David MacKenzie began work on a new tool--a tool for automatically generating makefiles for a GNU project. MacKenzie's work on Automake lasted about a year during 1994, ending around November of that year. A year later, during November of 1995, Tom Tromey (of RedHat and Cygnus fame) took over development of the Automake project. Tromey really had very much a defining role in Automake. In fact, although MacKenzie wrote the initial version of Automake in Bourne shell script, Tromey completely rewrote the tool in Perl over the following year. Tromey continued to maintain and enhance Automake during the next 5 years.

NOTE: Do not confuse the requirements of Automake on the project maintainer with the requirements of a generated build system on the end user. Perl is required by Automake, not by the generated build system.
Around February of 2000, Alexandre Duret-Lutz began to take a more active role in the development of the Automake project, and by the end of that year, had pretty much taken over project maintenance. Duret-Lutz's role as project lead lasted until about mid-2007. Since then, the project has been maintained by Eric Blake of the Free Software Foundation (FSF), with strong leadership (and most of the repository check-in's, for that matter) from automake mailing list contemporaries such as Ralf Wildenhues and Akim Demaille. *(I owe many heartfelt thanks to Ralf for kindly answering so many seemingly trivial questions while I worked on this book.)*
Sometime early during development of the GNU Coding Standards (GCS), it became clear to MacKenzie that much of a GNU project makefile was fairly boilerplate in nature. This is because the GCS guidelines are fairly specific about how and where a project's products should be built, tested, and installed. These conditions have allowed Automake syntax to be concise--in fact, it's terse, almost to a fault. One Automake statement represents a *lot* of functionality. The nice thing, however, is that once you understand it, you can get a fairly complete, complex and functionally correct build system up and running in short order--I mean on the order of minutes, not hours or days.

Getting down to business

Let's face it, writing a makefile is hard. Oh, the initial writing is fairly simple, but getting it right is often very difficult--the devil, as they say, is in the details. Like any high-level programming language, make syntax is often conducive to formulaic composition. That's just a fancy way of saying that once you've solved a "make problem", you're inclined to memorize the solution and apply the same formula the next time that problem crops up--which happens quite often when writing build systems.

So what advantages does Automake give us over our hand-coded Makefile.in templates, anyway? Well, that's pretty easy to answer with a short example. Consider the following changes to the files in our project directory structure (these commands are executed from jupiter's top-level directory):

```
$ rm autogen.sh Makefile.in src/Makefile.in

$ echo "SUBDIRS = src" > Makefile.am

$ echo "bin_PROGRAMS = jupiter

> jupiter_SOURCES = main.c" > src/Makefile.am

$ touch NEWS README AUTHORS ChangeLog

$ vi configure.ac

...

AC_INIT([Jupiter], 1.0, [bugs@jupiter.org])

AM_INIT_AUTOMAKE

AC_CONFIG_SRCDIR([src/main.c])

...

$ autoreconf -i

$
```

The "`rm`" command deletes our hand-coded Makefile.in templates and the autogen.sh script we wrote to ensure that all the support scripts and files were copied into the root of our project directory. We won't be needing this script anymore because we're upgrading jupiter to Automake proper.

For the sake of brevity in the text, I used `echo` statements to write the new Makefile.am files, but you may, of course, use an editor if you wish. *NOTE: There is a hard carriage-return after "`bin_PROGRAMS = jupiter`" in the third line. The shell will continue to accept input after the carriage return until the quotation is closed on the following line.*

The `touch` command is used to create new empty versions of the `NEWS`, `README`, `AUTHORS` and `ChangeLog` files in the project root directory. These files are required by the GCS for all GNU projects. While they're *not* required for non-GNU programs, they've become something of an institution in the FOSS world--you'd do well to have these files, properly formatted, in your project, as users have come to expect them. The GCS document covers the format and contents of these files. Section 6 covers the `NEWS` and `ChangeLog` files, and Section 7 covers the `README` and `INSTALL` files. The `AUTHORS` file is a list of people (names and optional email addresses) to whom attribution should be given.

Enabling Automake in `configure.ac`

Finally, I've added a single line to the configure.ac file, `AM_INIT_AUTOMAKE` between the `AC_INIT` and `AC_CONFIG_SRCDIR` statements. Besides the normal requirements of an

Autoconf input file, this is the *only* line that's required to enable Automake in a project that's already configured with Autoconf. The AM_INIT_AUTOMAKE macro accepts an optional argument--a white-space separated list of option tags, which can be passed into this macro to modify the general behavior of Automake. The following is a comprehensive list of options for Automake version 1.10:

- `gnits`
- `gnu`
- `foreign`
- `cygnus`
- `ansi2knr`
- `path/ansi2knr`
- `check-news`
- `dejagnu`
- `dist-bzip2`
- `dist-lzma`
- `dist-shar`
- `dist-zip`
- `dist-tarZ`
- `filename-length-max=99`
- `no-define`
- `no-dependencies`
- `no-dist`
- `no-dist-gzip`
- `no-exeext`
- `no-installinfo`
- `no-installman`
- `nostdinc`
- `no-texinfo.tex`
- `readme-alpha`
- `std-options`
- `subdir-objects`
- `tar-v7`
- `tar-ustar`
- `tar-pax`
- `<version>`
- `-W<category>`
- `--warnings=<category>`

I won't spend a lot of time on the option tag list at this point. For a detailed description of each option, check out Chapter 17 of the GNU Automake manual. I will, however, point out a few of the most useful options.

The `check-news` option will cause "make dist" to fail if the current version doesn't show up in the first few lines of the `NEWS` file. The `dist-*` tags can be used to change the default distribution package type. Now, these are handy because often developers want to distribute tar.bz2 files, rather than tar.gz files. By default, "make dist" builds a tar.gz file. You can override this by using "make dist-bzip2", but this is more painful than it needs to be for projects that like to use bzip2 by default. The `readme-alpha` option can be used to temporarily alter the behavior of the build and distribution process during alpha releases of a project. First, a file named `README-alpha`, found in the project root directory, will be distributed automatically while using this option. This option will also alter the expected versioning scheme of the project.

The `<version>` option is actually a placeholder for a numeric version number. This value represents the lowest version number of Automake that is acceptable for this project. For instance, if `1.10` is passed as a tag, then Automake will fail if it's version is less than 1.10. The `-W<category>` and `--warnings=<category>` options indicate that the project would like to use Automake with various warning categories enabled.

What we get from Automake

The last line of the example executes the `autoreconf -i` command, which, as I've already discussed in prior chapters, regenerates all Autotools-generated files according to the configure.ac file. This time, with the inclusion of the `AM_INIT_AUTOMAKE` statement, the `-i` option properly tells Automake to add any missing files. The `-i` option need only be used once in a newly checked out work area. Once the missing utility files have been added, the `-i` option may be dropped.

These few commands create for us an Automake-based build system containing everything that we wrote into our original Makefile.in templates, *except that this one is more correct and functionally complete.* A quick glance at the resulting generated Makefile.in template shows us that, from just a couple of input lines, Automake has done a significant amount of work for you. The resulting top-level Makefile.in template (remember, the configure script turns these templates into Makefiles), is nearly 18K in size. The original files were only a few hundred bytes long.

A generated Automake build system supports the following important `make` targets--and this list is *not* comprehensive:

- `all`
- `distdir`
- `install`
- `install-strip`
- `install-data`
- `install-exec`
- `uninstall`
- `install-dvi`
- `install-html`
- `install-info`
- `install-ps`
- `install-pdf`
- `installdirs`
- `check`
- `installcheck`
- `mostlyclean`
- `clean`
- `distclean`
- `maintainer-clean`
- `dvi`
- `pdf`
- `ps`
- `info`
- `html`
- `tags`
- `ctags`
- `dist`
- `dist-bzip2`
- `dist-gzip`
- `dist-lzma`
- `dist-shar`
- `dist-zip`
- `dist-tarZ`
- `uninstall`

As you can see, this goes a bit beyond what was provided in your hand-coded Makefile.in templates. And Automake writes all of this functionality automatically, correctly and quickly for each project that you instrument in the manner outlined above.

So, what's in a Makefile.am file?

You'll no doubt recall from Chapter 3 that Autoconf accepts shell script, sprinkled with M4 macros, and generates the same shell script with those macros fully expanded into additional shell script. Likewise, Automake accepts as input a makefile, sprinkled with Automake commands. As with Autoconf, the significance of this statement is that Automake input files are nothing more or less than makefiles with additional syntax.

One very significant difference between Autoconf and Automake is that Autoconf generates *no* output text except for the existing shell script in the input file, plus any additional shell script resulting from the expansion of embedded M4 macros. Automake, on the other hand, assumes that all makefiles should contain a minimal infrastructure designed to support the GCS, in addition to any targets and variables that you specify. To illustrate this point, I'll create a `temp` directory in the root of the jupiter project, and add an *empty* Makefile.am file to that directory. Then I'll add this new Makefile.am to my project, like this:

```
$ mkdir temp

$ touch temp/Makefile.am

$ echo "SUBDIRS = src temp" > Makefile.am

$ vi configure.ac

...

AC_CONFIG_FILES([Makefile

                 src/Makefile

                 temp/Makefile])

...

$ autoreconf

$ ./configure

...

$ ls -1sh temp

total 20K

 12K Makefile

   0 Makefile.am

8.0K Makefile.in
```

```
$
```

Thus we can see that Automake considers a certain amount of support code to be indispensable in every makefile. Even with an empty Makefile.am file, you end up with about 12K of code in the resulting Makefile, which is generated by configure (config.status) from an 8K Makefile.in template. Incidentally, it's fairly instructive to examine the contents of this Makefile.in template to see the Autoconf substitution variables that are passed in, as well as the framework code that Automake generates.

Since the make utility uses a fairly rigid set of rules for processing makefiles, Automake takes some minor "literary license" with your additional make code. Specifically, two basic rules are followed by Automake when generating Makefile.in templates from Makefile.am files that contain additional non-Automake-specific syntax (rules, variables, etc):

- Make *variables* that you define in your Makefile.am files are placed at the top of the resulting Makefile.in template, immediately following any Automake-generated variable definitions.
- Make *rules* that you specify in your Makefile.am files are placed at the end of the resulting Makefile.in template, immediately following any Automake-generated rules. Make doesn't care where rules are located relative to one another, because it reads all of the rules and stores them in an internal database before processing any of them. Variables are treated in a similar manner. To prove this to yourself, try referencing a variable in a makefile before its definition. Make binds values to variable references at the last possible moment, right before command lines containing these references are passed to the shell for execution.

Often, you won't need to specify anything besides a few Automake commands within a given Makefile.am, but there are frequent occasions when you will want to add your own make targets. This is because, while Automake does a lot for you, it can't anticipate *everything* you might wish to do in your build system. It's in this "grey" area where most developers begin to complain about Automake.
I'll spend the rest of this chapter examining the functionality provided by Automake. Later, I'll get into some tricks you can use to significantly enhance existing Automake functionality.

Analyzing our new build system

I will now spend some time looking at what I put into those two simple Makefile.am files. I'll start with the top-level file, with its single line of Automake code:

`Makefile.am`

```
SUBDIRS = src
```

It's pretty easy to divine the primary purpose of this line of text just by looking at the text itself. It appears to be indicating that I have a sub-directory in our project called `src`. In fact, this line tells Automake several things about our project:

- There are one or more immediate sub-directories containing Makefile.am files to be processed, in addition to this file.
- Directories in this space-delimited list are to be processed in the order specified.
- Directories in this list are to be recursively processed for all primary make targets.
- Directories in this list are to be treated as part of the project distribution.

`SUBDIRS` is not just a make variable: it's recognized by Automake to have special meaning, besides the intrinsic meaning associated with common make variables. As you continue to study Automake constructs, this theme will come up over and over again. Most Automake statements are, in fact, just make variables with special meaning to Automake.

Another point about the `SUBDIRS` variable is that it may be used in an arbitrarily complex directory structure, to process Makefile.am files within a project. You might say that `SUBDIRS` is the "glue" that links Makefile.am files together in a project's directory hierarchy.

One final point about `SUBDIRS` is that the current directory is *implicitly listed last* in the `SUBDIRS` list, meaning that the current directory will be built *after* all of the directories listed in the `SUBDIRS` variable. You may change this implied ordering if you wish, by using "." (meaning the current directory) anywhere in the list. This is important because it's sometimes necessary to build the current directory before one or more subdirectories.

Let's move down a level now into the `src` directory. The `src/Makefile.am` file contains slightly more code for you to examine; two lines rather than one:

`src/Makefile.am`

```
bin_PROGRAMS = jupiter

jupiter_SOURCES = main.c
```

Primaries

The first line, "`bin_PROGRAMS = jupiter`" lists the products generated by this Makefile.am file. Multiple files may be listed in this variable definition, separated by white space. The variable name itself is made up of two parts, the installation location, `bin`, and the product type, `PROGRAMS`. GNU Automake documentation calls the product type portion of these variables a "primary". The following is a list of valid primaries for version 1.10 of Automake:

- `PROGRAMS`
- `LIBRARIES`
- `LISP`
- `PYTHON`
- `JAVA`
- `SCRIPTS`
- `DATA`
- `HEADERS`
- `MANS`
- `TEXINFOS`

NOTE: Libtool adds `LTLIBRARIES` to the primaries list supported by Automake. I'll examine this and other Automake extensions provided by Libtool in Chapter 5.

You could consider primaries to be "product classes", or types of products that might be generated by a build system. This being the case, it's pretty clear that not all product classes are handled by Automake. What differentiates one class of product from another? Basically differences in handling semantics during build and installation. `PROGRAMS`, for example are built using different compiler and linker commands than are `LIBRARIES`. Certainly `LISP`, `JAVA` and `PYTHON` products are handled differently--the build system uses entirely different tool chains to build these types of products. And `SCRIPTS`, `DATA` and `HEADERS` aren't generally even built (although they might be), but rather simply copied into appropriate installation directories.

`PROGRAMS` also have different execution, and thus installation, semantics from `LISP`, `PYTHON` and `JAVA` programs. Products that fit into the `PROGRAMS` category are generally executable by themselves, while `LISP`, `JAVA` and `PYTHON` programs require virtual machines and interpreters.

What makes this set of primaries important? The fact that they cover 99 percent of the products created in official GNU projects. If your project generates a set of products that define their own product class, or use a product class not listed in this set of primaries, then you might do well to simply stick with Autoconf until support is added to Automake for your product class. Another option is to add support yourself to Autoconf for your product class, but doing so requires a deep knowledge of both the product class and the Automake Perl script. I believe it's fair to say, however, that this set of primaries covers a wide range of currently popular product classes.

Prefixes

Supported installation locations are provided by the GCS document. This is the same list that I provided to you in Chapter 2. I'll relist them here for convenience:

- `bindir`
- `sbindir`
- `libexecdir`
- `datarootdir`
- `datadir`
- `sysconfdir`
- `sharedstatedir`
- `localstatedir`
- `includedir`
- `oldincludedir`
- `docdir`
- `infodir`
- `htmldir`
- `dvidir`
- `pdfdir`
- `psdir`
- `libdir`
- `lispdir`
- `localedir`
- `mandir`
- `manNdir`

You may have noticed that I left a few entries out of this version of the list. Essentially, all entries ending in `dir` are viable prefixes for Automake primaries. Besides these standard GCS installation locations, three other installation locations are defined by Automake to have enhanced meaning:

- `pkglibdir`
- `pkgincludedir`
- `pkgdatadir`

The `pkg` versions of the `libdir`, `includedir` and `datadir` prefixes are designed to install products into subdirectories of these installation locations that are named after the package. For example, for the jupiter project, the `pkglibdir` installation location would be found in `$(exec-prefix)/lib/jupiter`, rather than the usual `$(exec-prefix)/lib` directory.

If this list of installation locations isn't comprehensive enough, don't worry--Automake provides a mechanism for you to define your own installation directory prefixes. Any make variable you define in your Makefile.am file that ends in `dir` can be used as a valid primary prefix. To reuse the example found in the GNU Automake manual, let's say you wish to install a set of XML files into an `xml` directory within the system data directory. You might use this code to do so:

```
xmldir = $(datadir)/xml

xml_DATA = file1.xml file2.xml file3.xml ...
```

Note that the same naming conventions are used with custom installation locations as with the standard locations. Namely, that the variable ends with `dir`, but the `dir` portion of the variable name is left off when using it as a primary prefix.

There are also several prefixes with special meanings not related to installation locations:

- `check`
- `noinst`
- `EXTRA`

The `check` prefix indicates products that are built only for testing purposes, and thus will not be installed at all. Products listed in primary variables that are prefixed with `check` aren't even built if the user never types `make check`.

The `noinst` prefix indicates that the listed products should be built, but not installed. For example, a static so-called "convenience" library might be built as an intermediate product, and used in other stages of the build process to build final products. Such libraries are not designed to be installed, so the prefix shouldn't be an installation location. The `noinst` prefix serves this purpose.

The `EXTRA` prefix is used to list programs that are conditionally built. This is a difficult concept to explain in a few paragraphs, but I'll give it a try. All product files must be listed statically (as opposed to being calculated at build-time) in order for Automake to generate a Makefile.in template that will work for any set of input commands. However, a project maintainer may elect to allow some products to be built conditionally, based on configuration options given to the configure script. If some products are listed in variables generated by the configure script, then these products should also be listed in a primary prefixed with "`EXTRA`", like this:

```
EXTRA_PROGRAMS = myoptionalprog

bin_PROGRAMS myprog $(optional_programs)
```

Here, it is assumed that the "`optional_programs`" variable is defined in the configure script, and listed in an `AC_SUBST` macro. This way, Automake can know in advance that "`myoptionalprog`" *may* be built, and so generate rules to build it. Any program that may or may not be built, based on configuration options should be specified in `EXTRA_PROGRAMS`, so that Automake can generate a makefile that *could* build it if requested to do so.

"Super" prefixes

Some primaries allow a sort of "super" prefix to be prepended to a prefix/PRIMARY variable specification. Such modifiers may be used together on the same variable where it makes sense. Thus, these "super" prefixes modify the normal behaviour of a prefix/PRIMARY specification. The existing modifiers include:

- `dist`
- `nodist`
- `nobase`

The `dist` modifier indicates a set of files that should be distributed (that is, included in the distribution package when "make dist" is executed). The `dist` modifier is used with files that are normally not distributed, but may be used explicitly anywhere for clarity. For instance, assuming that some source files for a product should be distributed, and some should not (perhaps they're generated), the following rules might be used:

```
dist_jupiter_SOURCES = file1.c file2.c

nodist_jupiter_SOURCES = file3.c file4.c
```

While the `dist` prefix is redundant in this example, it is nonetheless useful to the casual reader.

The `nobase` modifier is used to suppress the removal of path information from installed header files that are obtained from subdirectories by a Makefile.am file. For instance, assume that installable jupiter project header files exist in a subdirectory of the `src` directory "`jupiter`":

```
nobase_dist_include_HEADERS = \

  jupiter/jupiter_interface.h
```

Normally, such a header file would be installed into the `/usr(/local)/include` directory as simply `jupiter_interface.h`. However, if the `nobase` modifier is used, then the extra path information would not be removed, so the final resting place of the installed header would instead be `/usr(/local)/include/jupiter/jupiter_interface.h`.

Notice also in this example that I combined the use of the `nobase` modifier with that of the `dist` modifier--just to show the concept.

Product sources

The second line in `src/Makefile.am` is "`jupiter_SOURCES = main.c`". This variable lists the source files used to build the `jupiter` program. Like product variables made from prefixes and primaries, this type of variable is derived from two parts, the product name, `jupiter` in this case, and the dependent type. I call it the "dependent type" because this variable lists source files on which the product depends. Ultimately, Automake adds these files to make rule dependency lists.

The `EXTRA` prefix may also be used sometimes as a super prefix modifier. When used with a product SOURCES variable (eg., `jupiter_SOURCES`), `EXTRA` can be used to specify extra source files that may or may not be used, which are directly associated with the jupiter product:

```
EXTRA_jupiter_SOURCES = possibly.c
```

In this case, `possibly.c` may or may not be compiled--perhaps based on an AC_SUBST variable.

Unit tests - supporting "`make check`"

I mentioned earlier that this Automake-generated build system provided the same functionality as our hand-coded build system. Well, I wasn't completely truthful when I said that. For the most part, that was an accurate statement, but what's still missing is our simple-minded `make check` functionality. The `check` target is indeed supported by our new Automake build system, but it's just not hooked up to any real functionality. Let's do that now.

You'll recall in Chapter 2 that you added code to the `src/Makefile` to run the jupiter program and check for the proper output string when the user entered "make check". You did this with a fairly simple addition to our `src/Makefile`:

```
...

check: all

        ./jupiter | grep "Hello from .*jupiter!"

        @echo "*** ALL TESTS PASSED ***"

...
```

As it turns out, Automake has some solid support for unit tests. Unfortunately, the documentation consists of Chapter 15 of the GNU Automake manual--a single page of text--half of which is focused on the obscure DejaGNU test suite syntax. Nevertheless, adding unit tests to a Makefile.am file is fairly trivial. To add a simple "grep test" back into the new Automake-generated build system, I've added a few more lines to the bottom of the `src/Makefile.am` file:

`src/Makefile.am`

```
bin_PROGRAMS = jupiter

jupiter_SOURCES = main.c

jupiter_CPPFLAGS = -I$(top_srcdir)/common

jupiter_LDADD = ../common/libjupcommon.a



check_SCRIPTS = greptest.sh

TESTS = $(check_SCRIPTS)



greptest.sh:

        echo './jupiter | grep \

          "Hello from .*jupiter!"' > greptest.sh

        chmod +x greptest.sh



CLEANFILES = greptest.sh
```

The `check_SCRIPTS` line is clearly a prefixed primary. The `SCRIPT` primary indicates a "built" script, or a script that is somehow generated at build time. Since the prefix is "check", you know that scripts listed in this line will only be built when the user enters "make check" (or "make distcheck"). However, this is as far as Automake goes in supporting such built scripts with Automake-specific syntax. You must supply a make rule for building the script yourself.
Furthermore, since you supplied the rule to generate the script, you must also supply a rule for cleaning the file. Automake provides an extension to the generated `clean` rule, wherein all files listed in a special `CLEANFILES` variable are added to the list of automatically cleaned files.
The `TESTS` line is the important one here, in that it indicates which targets are built and executed when a user enters "make check". Since the "`check_SCRIPTS`" variable contains a complete list of these targets, I simply reused its value here.
Generating scripts or data files in this manner is a very useful technique. I'll present some more interesting ways of doing this sort of thing in Chapter 8.

Adding complexity with convenience libraries

Well, jupiter is fairly trivial, as free software projects go. In order to highlight some more of the key features of Automake, I'm going to have to expand jupiter into something a little bit more complex (if not functional).

I'll start by adding a convenience library, and having jupiter consume this library. Essentially, I'll move the code in main.c to a library source file, and then call the function in the library from jupiter's main routine. Start with the following commands, executed from the top-level project directory:

```
$ mkdir common

$ touch common/jupcommon.h

$ touch common/print.c

$ touch common/Makefile.am
```

Add the following text to the .h and .c files:

`common/jupcommon.h`

```
int print_routine(char * name);
```

`common/print.c`

```
#include <jupcommon.h>



#if HAVE_CONFIG_H
```

```
# include <config.h>
#endif

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

#if HAVE_PTHREAD_H
# include <pthread.h>
#endif

static void * print_it(void * data)
{
   printf("Hello from %s!\n", (char *)data);
   return 0;
}

int print_routine(char * name)
{
#if ASYNC_EXEC
   pthread_t tid;
   pthread_create(&tid, 0, print_it, name);
   pthread_join(tid, 0);
#else
   print_it(name);
#endif
   return 0;
```

```
}
```

As promised, `print.c` is merely a copy of `main.c`, with a couple of small modifications. First, I renamed `main` to `print_routine`, and second, I added the inclusion of the jupcommon.h header file at the top. This header file (as you can see) merely provides `print_routine`'s prototype to the new `src/main.c`, where it's called from `main`. Modify `src/main.c` to look like this:

`src/main.c`

```
#include <jupcommon.h>



int main(int argc, char * argv[])

{

   print_routine(argv[0]);

   return 0;

}
```

And now for the new `common/Makefile.am` file; add the following text to this file:

`common/Makefile.am`

```
noinst_LIBRARIES = libjupcommon.a

libjupcommon_a_SOURCES = jupcommon.h print.c
```

Let's take a look at this file for a minute. You'll recall from our discussion of Automake primaries and prefixes that the first line indicates the products to be built and installed by this Makefile.am file. In this case, the `noinst` prefix indicates that this library should not be installed at all. This is because you're creating a "convenience" library, or a library designed solely to make using the source code in the common directory more convenient for two or more consumers. (Granted, you only have one consumer at this point--the jupiter program--but later on you'll add another consumer of this library, and then it will make more sense.)

The library we're creating will be called "libjupcommon.a"--this is a static library, also known as an "object archive". Object archives are merely packages containing one or more object (.o) files. They can't be executed, or loaded into a process address space, as can shared libraries. They can only be added to a linker command line. The linker is smart enough to realize that such archives are merely groups of object files. The linker extracts the object files it needs to complete the linkage process when building a program or shared library.

The second line represents the list of source files associated with this library. I chose to place both the header and the C source file in this list. I could have chosen to use a "`noinst_HEADERS`" line for the header file, but it was unnecessary because the "`libjupcommon_a_SOURCES`" list works just as well. The appropriate time to use "`noinst_HEADERS`" is when you have a directory that contains no source (.c) files--such as an internal include directory. Personally, I don't care for this style of project directory structure organization. I prefer to place private header files right along side of the source code they represent. As a result, I never seem to need "`noinst_HEADERS`" in my projects.

Notice the format of the "`libjupcommon_a_SOURCES`" variable. Automake transforms library and program names in the product list into derived variable names by converting all characters except for letters, numbers and at-signs (`@`) into underscore characters. Thus, a library named `libc++.a` generates a SOURCES variables called `libc___a_SOURCES` (there are three consecutive underscores in that variable name).

Clean up your top-level project directory, removing all files and directories except those that we've written by hand so far. Also remove all Makefile.in files in the top-level directory and in sub-directories. The top-level directory should look like this when you're done:

```
$ ls -1F

AUTHORS

ChangeLog

common/

configure.ac

COPYING

INSTALL

src/

Makefile.am

NEWS

README
```

Edit the `SUBDIRS` variable in the top-level Makefile.am file to include the new common directory that we just added:

`Makefile.am`

```
SUBDIRS = common src
```

Now you have to add some additional information to the `src/Makefile.am` file so that the generated Makefile can find the new library and header file you created in the common directory. Add two more lines to the end of the existing file, in this manner:

`src/Makefile.am`

```
bin_PROGRAMS = jupiter
```

```
jupiter_SOURCES = main.c

jupiter_CPPFLAGS = -I$(top_srcdir)/common

jupiter_LDADD = ../common/libjupcommon.a
```

Like the `jupiter_SOURCES` variable, these two new variables are obviously derived from the program name. The `jupiter_CPPFLAGS` variable is used to add product-specific C preprocessor flags to the compiler command line for all source files that are built for the jupiter program. The `jupiter_LDADD` variable is used to add libraries to the linker command line for the jupiter program.

These product-specific option variables are used to pass options to the compiler and linker command lines. The option variables currently supported by Automake for programs include:

- `program_CCASFLAGS`
- `program_CFLAGS`
- `program_CPPFLAGS`
- `program_CXXFLAGS`
- `program_FFLAGS`
- `program_GCJFLAGS`
- `program_LFLAGS`
- `program_OBJCFLAGS`
- `program_RFLAGS`
- `program_UPCFLAGS`
- `program_YFLAGS`

For static library products use `library_LIBADD`, instead of `program_LDADD`. The _LIBADD variable for libraries allows you to specify additional object files and static libraries that should be added to the static archive you're currently building. This can be handy for combining multiple convenience libraries. Consider the difference between these cases: The `library_LIBADD` variable is merely allowing you to specify already built objects--either libraries or actual object modules--to the library you're currently building. This can't be accomplished with the `library_SOURCES` variable, because `library_SOURCES` members are compiled, whereas `library_LIBADD` members are already built.

Additionally, the `program_LDADD` variable generally expects linker command line options such as `-lz` (to add the libz library to the linker's library specification for this program), while the `library_LIBADD` variable is formatted as a list of fully specified objects (eg., libabc.a file1.o). This rule isn't particularly strict however, as I'll explain shortly here. Quite frankly, it doesn't really matter, as long as the final command line composed by Automake from all of these variables makes sense to the linker.

File-level option variables

Often you'll see unprefixed variables like `AM_CPPFLAGS` or `AM_LDFLAGS` used in a Makefile.am. This is the per-file form of these flags, rather than the per-product form. The per-file forms are used when the developer wants the same set of flags to be used for all products within a given Makefile.am file.

Sometimes you need to set a group of preprocessor flags for all products in a Makefile.am file, but add additional flags for one particular target. When you use a per-product flag variable, you need to include the per-file variable explicitly, like this:

```
AM_CFLAGS = ... some flags ...
```

```
program_CFLAGS = ... more flags ... $(AM_CFLAGS)
```

User variables, such as `CFLAGS`, should never be modified by configuration scripts or makefiles. These are reserved for the end-user, and will be always be appended to the per-file or per-product versions of these variables.

Regarding the `jupiter_LDADD` variable, `../common/libjupcommon.a` merely adds an object to the linker command line, so that code in this library may become part of the final program. Note that this sort of syntax is really only necessary for libraries built as part of your own package. If you're linking your program with a library that's installed on the user's system, then the configure script should have found it, and automatically added an appropriate reference to the linker's command line.

In the `jupiter_CPPFLAGS` variable, the `-I$(top_srcdir)/common` directive tells the C preprocessor to add a search directory to its list of locations in which to look for header file references. Specifically, it indicates that header files referenced in C source files with angle brackets (`<` and `>`) should be searched for in this include search path. Header files referenced with double-quotes are not searched for, but merely expected to exist in the specified directory, relative to the directory containing the referencing source file.

Getting back to our example--edit the configure.ac file; add a reference to the AC_CONFIG_FILES macro for the new generated common/Makefile, in this manner:

`configure.ac`

```
...

AC_CONFIG_FILES([Makefile

                 common/Makefile

                 src/Makefile])

...
```

Okay, now give your updated build system a try. Add the `-i` option to the `autoreconf` command so that it will install any additional missing files that might be required after our enhancements:

```
$ autoreconf -i

configure.ac:6: installing `./missing'

configure.ac:6: installing `./install-sh'

common/Makefile.am:1: library used but `RANLIB'

   is undefined. The usual way to define

   `RANLIB' is to add `AC_PROG_RANLIB' to

   `configure.ac' and run `autoconf' again.

common/Makefile.am: installing `./depcomp'
```

```
src/Makefile.am:3: compiling `main.c' with
   per-target flags requires `AM_PROG_CC_C_O' in
   `configure.ac'
autoreconf: automake failed with exit status: 1
```

Well, it appears that you're still not done yet. Since you've added a new type of entity to our build system--static libraries--Automake (via autoreconf) tells you that you need to add a new macro to the configure.ac file. The AC_PROG_RANLIB macro is a standard program check macro, just like AC_PROG_YACC or AC_PROG_LEX. There's a lot of history behind the use of the ranlib utility on archive libraries. I won't get into whether it's still useful with respect to modern development tools. It seems however, that wherever you see it used in modern Makefiles, there's always a comment about running ranlib in order to "add karma" to the archive. You be the judge...

Additionally, you need to add the Automake macro, AM_PROG_CC_C_O, because this macro defines constructs in the resulting configure script that support the use of per-product flags, such as `jupiter_CPPFLAGS`. Add these two macros to your configure.ac script:

`configure.ac`

```
...
# Checks for programs.
AC_PROG_CC
AC_PROG_INSTALL
AC_PROG_RANLIB
AM_PROG_CC_C_O
...
```

Alright, once more then, but this time I'm adding the `--force` option, as well as the `-i` option to the autoreconf command line to keep it quiet about adding files that already exist. (This seems like a pointless option to me, because the entire purpose of the `-i` option is to add *missing* files, not to add *all* files that are required, regardless of whether they already exist, or not, and then complain if they do exist.):

```
$ autoreconf -i --force
configure.ac:15: installing `./compile'
```

Blessed day! It works. And it really wasn't too bad, was it? Automake told you exactly what you needed to do.

(I always find it ironic when a piece of software tells you how to fix your input file--why didn't it just do what it knew you wanted it to do, if it understood your intent without the correct syntax?! Okay, I understand the "purist" point of view, but why not just do "the right thing", with a side-line comment about your ill-formatted input text? Eventually, you'd be annoyed enough to fix the problem anyway, wouldn't you? Of course you would!)

A word about the utility scripts

It seems that Automake has added yet another missing file--the "compile" script is a wrapper around some older compilers that do not understand the use of both `-c` and `-o` on the command line to name the object file differently than the source file. When you use product-specific flags, Automake has to generate code that may compile source files multiple times with different flags for each file. Thus it has to name the files differently for each set of flags it uses. The requirement for the compile script actually comes from the inclusion of the AM_PROG_CC_C_O macro.
At this point, you have the following Autotools-added files in the root of our project directory structure:

- `compile`
- `depcomp`
- `install-sh`
- `missing`

These are all scripts that are executed by the configure script, and by the generated Makefiles at various points during the end-user build process. Thus, the end-user will need these files.You can only get these files from Autotools. Since the user shouldn't be required to have Autotools installed on the final target host, you need to make these files available to the user somehow.

These scripts are automatically added (by "make dist") to the distribution tarball. So, do you check them in to the repository, or not? The answer to this question is debatable, but generally I recommend against doing this. Anyone who will be creating a distribution tarball should also have the Autotools installed, and should be working from a repository work area. As a result, this maintainer will also be running `autoreconf -i (--force)` to ensure that she has the latest updated Autotools-provided utility scripts. Checking them in will only make it more probable that they become out of date as time goes by.
As mentioned in Chapter 2, this sentiment goes for the configure script as well. Some people argue that checking the utility and configure scripts into the project repository is beneficial, because it ensures that someone checking out a work area can build the project from the work area without having the Autotools installed. But is this really important? Shouldn't developers and maintainers be expected to have more advanced tools? My personal philosophy is that they should. Yours may differ. Occasionally, an end user will need to build a project from a work area, but this should be the exceptional case, not the typical case. If it is the typical case, then there are bigger problems with the project than can be solved in this discussion.

What goes in a distribution?

In general, Automake determines automatically what should go into a distribution created with `make dist`. This is because Automake is vary aware of every single file in the build process, and what it's used for. Thus, it need not be told explicitly which files should be in the package, and which should be left behind.
An important concept to remember is that Automake wants to know statically about every source file used to build a product, and about every file that's installed. This means, of course, that all of these files must somehow be specified at some point in a Makefile.am

primary variable. This bothers some developers--and with good reason. There are cases where dozens of installable files are generated by tools using long, apparently random and generally unimportant naming conventions. Listing such generated files statically in a primary variable is problematic, to say the least.

I'll cover techniques that can be used to work around such problem cases later in this book. At this point, however, I'd like to introduce the `EXTRA_DIST` variable for those cases where file system entities are not part of the Automake build process, but should be distributed with a distribution tarball. The `EXTRA_DIST` variable contains a space-delimited list of files and directories which should be added to the distribution package when "make dist" is executed.

```
EXTRA_DIST = windows
```

This might be used to add, for example, a `windows` build directory to the distribution package. Such a directory would be otherwise ignored by Automake, and then your windows users would be upset when they unpacked your latest tarball. Note in this example that `windows` is a directory, not a file. Automake will automatically and recursively add every file in this directory to the distribution package.

Summary

In this chapter, I've covered a fair number of details about how to instrument a project for Automake. The project I chose to instrument happened to already be instrumented for Autoconf, which is the most likely scenario, as you'll probably be adding Autoconf functionality to your bare projects first in most cases.

What I've explicitly *not* covered are situations where you need to extend Automake to handle your special cases, although I've hinted at this sort of thing from time to time. In the next chapter, I'll examine adding Libtool to the jupiter project, and then in Chapter 6, I'll Autotool-ize a real-world project, consisting of several hundred source files and a custom build system that takes the form of a GNU makefile designed to use native compilers on multiple platforms including Solaris, AIX, Linux, Mac OS and Windows, among others. I'll warn you up front thatI'll be remaining true to the original mission statement of this book in that we'll not be trying to get Autotools to build Microsoft Windows products.

Source archive

Download the attached source archive for the original sources associated with this chapter.

‹ Chapter 3: Configuring your project with Autoconf up Chapter 5: Building shared libraries with Libtool ›

Chapter 5: Building shared libraries with Libtool

Wed, 2008-07-09 19:23 -- John Calcote

The person who invented the concept of shared libraries should be given a raise... and a bonus. The person who decided that shared library management and naming conventions should be left to the implementation should be flogged.

This opinion is the result of too much negative experience on my part with building shared libraries for multiple platforms *without* the aid of Libtool. The very existence of Libtool stands as a witness to the truth of this sentiment.
Libtool exists for one purpose only--to provide a standardized, abstract interface for developers desiring to create portable shared libraries. It abstracts both the shared library build process, and the programming interfaces used to dynamically load and access shared libraries at run time.

This chapter has downloads!
Before I get into a discussion of the proper use of Libtool, I should probably spend a few minutes on the features and functionality provided by shared libraries, so that you will understand the scope of the material I'm covering here.

The benefits of shared libraries

Shared libraries provide a way to ship reusable chunks of functionality in a convenient package that can be loaded into a process address space, either automatically at program load time by the operating system loader, or by code in the application itself, when it decides to load and access the library's functionality. The point at which an application binds functionality from a shared library is very flexible, and determined by the developer, based on the design of the program and the needs of the end-user.

The interfaces between the program executable and modules defined as shared libraries must be well-designed by virtue of the fact that shared library interfaces must be well-specified. This rigorous specification promotes good design practices. When you use shared libraries, you're essentially forced to be a better programmer.

Shared libraries may be (as the name implies) shared among processes. This sharing is very literal. The code segments for a shared library can be loaded once into physical memory pages. Those same memory pages can then be mapped into the process address spaces for multiple programs. The data pages must, of course, be unique per process, but global data segments are often small compared to the code segments of a shared library. This is true efficiency.

Shared libraries are easily updated during program upgrades. The base program may not have changed at all between two revisions of a software package. A new version of a shared library may be laid down on top of the old version, as long as its interfaces have not been changed. When interfaces *are* changed, two versions of the same shared library may co-exist side-by-side, because the versioning scheme used by shared libraries (and supported by Libtool) allows the library files to be named differently, but treated as the same library. Older programs may continue to use older versions of the library, while newer programs may use the newer versions.
If a software package specifies a well-defined "plug-in" interface, then shared libraries can be used to implement user-configurable loadable functionality. This means that

additional functionality can become available to a program *after* it's been released, and third-parties can even add functionality to your program, if you publish a document describing your plug-in interface specification.
There are a few widely-known examples of systems such as this. Eclipse, for instance, is almost a pure plug-in framework. The base executable supports little more than a well-defined plug-in interface. Most of the functionality in an Eclipse application comes from library functions. Granted, Eclipse is written in Java, and uses Java class libraries, but the same concept can be (and has been) easily implemented in C or C++ using shared libraries.

How shared libraries work

As I mentioned above, the way a POSIX-based operating system implements shared libraries varies from platform to platform, but the general idea is the same for all platforms. The following discussion applies to shared library references that are resolved by the linker while the program is being built, and by the operating system loader at program load time.

Dynamic linking at load time

As a program executable image is being built, the linker (formally called a "link editor") maintains a table of unresolved function entry points and global data references. Each new symbol referenced by the object code being linked together, is added to this table. At the end of the linking process, all object files containing only unreferenced symbols are removed from the link list. All object files containing referenced symbols are linked together, and become part of the program executable image. If there are any outstanding references in the symbol table after all of the object files have been analyzed in this manner, the linker exits with an error message. On success, the final executable image may then be loaded and executed by a user. It is entirely self-contained, depending only upon itself.

Assuming that all undefined references are resolved during the linking process, if the list of objects to be linked contains one or more shared libraries, the linker will build the executable image from all *non-shared* objects specified on the linker command line. This includes all individual `.o` files and all static library archives. However it will add two tables to the binary image header; the first is the table of outstanding external references--those found only in shared libraries, and the second is a table of shared library names and versions in which the outstanding undefined references were found.
Later, when the operating system loader attempts to load this program, it must resolve the remaining outstanding references to symbols imported from the shared libraries named in the executable header. If the loader can't resolve all of the references, then a load error occurs, and the process is terminated with an operating system error message.

Note here that these external symbols are not tied to a *specific* shared library. The operating system will stop loading shared libraries as soon as it is able to resolve all of the outstanding symbol references. Usually, this happens after the last indicated shared library is loaded into the process address space, but there are exceptions.
NOTE: This process differs a bit from the way a Windows operating system resolves symbols in Dynamic Link Libraries (DLLs). On Windows, a particular symbol is tied by the linker at program build time to a specifically named DLL.
Using free-floating external references has both pros and cons. On some operating systems, unbound symbols can be satisfied by a library specified by the user. That is, a user can entirely *replace* a library (or a portion of a library) at run time by simply preloading one that contains the same symbols. On BSD and Linux based systems, for example, a user can use the "`LD_PRELOAD`" environment variable to inject a shared

library into a process address space. Since such libraries are loaded first by the loader before any other libraries, symbols in the preloaded libraries will be located first by the loader when it tries to resolve external references.

In the following example, the "`df`" utility is executed in an environment containing the `LD_PRELOAD` variable, set to a path referring to a library that presumably contains a heap manager. This technique can be used to debug problems in your programs. By preloading your own heap manager, you can capture memory leaks in a log file, or debug memory block overruns. This sort of technique is used by such widely-known debugging aids as the `valgrind` package.

```
$ LD_PRELOAD=~/lib/libmymalloc.so /bin/df

...
```

Unfortunately, free-floating symbols can also lead to problems. For instance, two libraries can provide the same symbol name, and the dynamic loader can inadvertently bind an executable to a symbol from the wrong library. At best, this will cause a program crash when the wrong arguments are passed to the mis-matched function. At worst, it can present security risks, because the mis-matched function might be used to capture passwords and security credentials passed by the unsuspecting program.

C-language symbols do not include parameter information, so it's rather likely that symbols will clash in this manner. C++ symbols are a bit safer, in that the entire function signature (minus the return type) is encoded into the symbol name. However, even C++ is not immune to hackers purposely replacing security functions with their own versions of those functions.

Automatic dynamic linking at run time

The operating system loader can also use a very late form of binding, often referred to as "lazy binding". In this situation, the external reference entries in the jump table in the program header are initialized such that they refer to code in the dynamic loader itself.

When a program first calls such a "lazy" entry, the call will be routed to the loader, which will then (potentially) load the proper shared library, determine the actual address of the function, reset the entry point in the jump table, and finally redirect to the (now available) shared library function. The next time this happens, the jump table entry will have been correctly initialized, and the program will jump directly to the called function.

This lazy binding mechanism makes for very fast program startup, because shared libraries whose symbols are not bound until they're needed aren't even loaded until they're first referenced by the application program. Now, consider this--they may *never* be referenced. Which means they may never be loaded, saving both time and space. An example of this situation might be a word processor with a thesaurus feature, implemented in a shared library. How often do you use your thesaurus? Using automatic dynamic linking, chances are that the shared library containing the thesaurus code will never be loaded in a given execution of your word processor.

The problems with this method should be obvious, at this point. While using automatic run-time dynamic linking can give you faster load times, and better performance and space efficiency, it can also cause abrupt terminations of your application--without warning. If the loader can't find the requested symbol--perhaps the required library is missing--then it has no recourse except to abort the process.

Why not ensure that all symbols exist when the program is loaded? Well, if the loader resolved all symbols at load time, then it might as well populate the jump table entries at that point. After all, it had to load all the libraries to ensure that the symbols actually exist. This then entirely defeats the purpose of this binding method. Furthermore, even if the loader did bother to check out all external references at the point when the program was first started, there's nothing to stop someone from deleting one or more of these libraries before it's used, while the program is still running. Thus, even the pre-check is defeated.

The moral of this story is that you get what you pay for. If you don't want to pay the insurance premium for longer up-front load times, and more space consumed (even if you may never really need it), then you may have to take the hit of a missing symbol at run time, causing a program crash.

Manual dynamic linking at run time

One possible solution to the aforementioned problem is to take personal responsibility for the work done by the system loader. Then, when things don't go right, you have a little more control over the outcome. In the case of the thesaurus module, was it really necessary to terminate the program if the thesaurus library could not be loaded or didn't provide the correct symbols? Of course not, but the loader didn't know that. Only the programmer can make such value judgements.

When a program manages dynamic linking manually at run-time, the linker is left entirely out of the equation. The program doesn't call any shared library functions directly. Rather, shared library functions are referenced though function pointers that are populated by the application program itself at run time.

The way this works is that a program calls an operating system function to manually load a shared library into its own process address space. This system function returns a "handle", or an opaque value representing the loaded library. The program then calls another loader function to import a symbol from the library referred to by the handle. If all goes well, the operating system returns the address of the requested function or data item in the desired library. The program may then call the function, or access the global data item through this pointer.

If something goes wrong in one of these two steps--say the library could not be found, or the symbol was not found within the library, then it becomes the responsibility of the program to define the results--perhaps display an error message, indicating that the program was not configured correctly.

This is a little nicer than the way automatic dynamic run-time linking works; while the loader has no option but to abort, the application has a higher-level perspective, and can handle the problem much more gracefully. The drawback, of course, is that you as the programmer have to manage the process of loading libraries and importing symbols within your application code. However, this process is not really very difficult, as I'll explain later in this chapter.

Using Libtool

An entire book could be written about the details of shared libraries and their implementations on various systems. This short primer will suffice for your immediate needs; so I'll move on to how Libtool can be used to make a package maintainer's life a little easier.

The Libtool project was started in 1996 by Gordon Matzigkeit. Libtool was designed to extend Automake, but can be used independently within hand-coded makefiles, as well. The Libtool project is currently maintained by Bob Friesenhahn, Peter O'Gorman, Gary Vaughan and Ralf Wildenhues.

Abstracting the build process

First, I'll look at how Libtool helps during the build process. Libtool provides a script (`ltmain.sh`) that `config.status` executes in a Libtool-enabled project. The `ltmain.sh` script builds a custom version of the `libtool` script, specifically for your package. This `libtool` script is then used by your project's makefiles to build shared libraries specified using the `LTLIBRARIES` primary. The `libtool` script is really just a fancy wrapper around the compiler, linker and other tools. The `ltmain.sh` script should be shipped in a distribution tarball, as part of your end-user build system. Automake-generated rules ensure that this happens properly.

The `libtool` script insulates the build system author from the nuances of building shared libraries on multiple platforms. This script accepts a well-defined set of options, converting them to appropriate platform- and linker-specific options on the target platform and tool set. Thus, the maintainer need not worry about the specifics of building shared libraries on each platform. She need only understand the available `libtool` script options. These are well specified in the GNU Libtool manual, and I'll cover many of them in this chapter.

On systems that don't support shared libraries at all, the `libtool` script uses appropriate commands and options to build and link static libraries. This is all done in such a way that the maintainer is isolated from the differences between building shared libraries and static libraries.

You can emulate building your package on a static-only system by using the "`--disable-shared`" option on the `configure` command line for your project. This causes Libtool to assume that shared libraries cannot be built on the target system.

Abstraction at run-time

Libtool can also be used to abstract the programming interfaces supplied by the operating system for loading libraries and importing symbols. Programmers who've ever dynamically loaded a library on a Linux system are familiar with the standard Linux shared library API, including the functions, `dlopen`, `dlsym` and `dlclose`. These functions are provided by a system-level shared library, usually named "`dl`". Unfortunately, not all POSIX systems that support shared libraries provide the `dl` library, or functions using these names.

To address these differences, Libtool provides a shared library called "`ltdl`", which provides a clean, portable library management interface, very similar to the `dlopen` interface provided by the Linux loader. The use of this library is optional, of course, but highly recommended, because it provides more than just a common API across shared library platforms. It also provides an abstraction for manual run-time dynamic linking between shared library and non-shared library platforms.

"What!? How can that work?" You might ask. On systems that don't provide shared libraries, Libtool actually creates internal symbol tables within the executable containing all of the symbols that would otherwise be found in shared libraries on systems that support shared libraries. By using these symbol tables on these platforms, the `lt_dlopen` and `lt_dlsym` functions can make your code appear to be loading and importing symbols, when in fact, the "load" function does nothing more than return a handle to the appropriate symbol table, and the "import" function returns the address of some code that's been statically linked into the program itself.

The `ltdl` library is, of course, not really necessary for packages that don't use manual run-time dynamic linking. But if your package does--perhaps by providing a plug-in interface of some sort--then you'd be well-advised to use the API provided by `ltdl` to manage loading and linking to your plug-in modules--even if you only target systems that provide good shared library services. Otherwise, your source code will have to consider the differences in shared library management between your many target platforms. At the very least, some of your users will have to put on their "developer" hats, and attempt to modify your code so that it works on their odd-ball platforms. (They may have to do so anyway, but when they finish, their work can then be incorporated into Libtool, so that everyone else can take advantage of their efforts.)

A word about the latest Libtool

The most current version of Libtool is 2.2. However, many popular GNU/Linux distributions are still shipping Libtool version 1.5, so many developers don't know about the changes between these two versions. The reason for this is that certain backward-compability issues were introduced after version 1.5 that make it difficult for GNU/Linux distros to support the latest version of Libtool. The upgrade probably won't happen until all (or almost all) of the packages they provide have updated their `configure.ac` scripts to properly use the latest version of Libtool.

This is somewhat of a "chicken-and-egg" scenario--if distros don't ship it, how will developers ever start using it on their own packages? So it's not likely to happen any time soon. If you want to make use of the latest Libtool version while developing your packages (and I highly recommend that you do so), you'll probably have to download, build and install it manually, or look for an updated Libtool package from your distribution provider.

Downloading, building and installing Libtool manually is really trival:

```
$ wget ftp.gnu.org/gnu/libtool/libtool-2.2.tar.gz

...

$ tar xzf libtool-2.2.tar.gz

$ cd libtool-2.2

$ ./configure && make

...

$ sudo make install

...
```

Be aware that the default installation location (as with most of the GNU packages) is `/usr/local`. If you wish to install it into the `/usr` hierarchy, then you'll need to use the `--prefix=/usr` option on the `configure` command line.

You might also wish to use the `--enable-ltdl-install` option on the configure command line to install the `ltdl` libraries and header files into your `lib` and `include` directories.

Adding shared libraries to Jupiter

Now that I've presented that background information, I will take a look at how I might add a Libtool shared library to the Jupiter project. First, consider what I might do with a shared library in Jupiter. As mentioned above, I might wish to provide my users with some library functionality that their own applications could use. I might also have several applications in my package that need to share the same functionality. A shared library is a great tool for both of these scenarios, because I get the benefits of code reuse and memory savings, as the cost of the memory used by shared code is amortized across multiple applications--both internal and external to my project.

I'll add a shared library to Jupiter that provides the print functionality I use in the `jupiter` application. I'll do this by having the new shared library call into the `libjupcommon.a` static library. Remember that calling a routine in a static library has the same effect as linking the object code for the called routine right into the calling application (or shared library, as the case may be). The called routine ultimately becomes an integral part of the calling binary image (program or shared library).

Additionally, I'll provide a public header file from the Jupiter project that will allow external applications to call this same functionality. By doing this, I can allow other applications to "display stuff" in the same way that the `jupiter` program "displays stuff". (This would be significantly cooler if I was actually doing something useful in `jupiter`!).

Using the `LTLIBRARIES` primary

Automake has built-in support for Libtool. The `LTLIBRARIES` primary is provided by code in the Automake package, not the Libtool package. This really doesn't qualify as a pure extension, but rather more of an add-on package for Automake, where Automake provides the necessary infrastructure for that specific add-on package. You can't access the `LTLIBRARIES` primary functionality provided by Automake without Libtool, because the use of this primary obviously generates make rules that call the `libtool` build script.

I state all of this here because it bothers me that you can't *really* extend the list of primaries supported by Automake without modifying the actual Automake source code. The fact that Automake is written in perl is somewhat of a boon, because it means that it's *possible* to do it. But you've *really* got to understand Automake source code in order to do it properly. I envision a future version of Automake whereby code may be added to an Automake extension file that will allow the dynamic definition of new primaries.

It's a bit like the old FOSS addage, generally offered to someone complaining about lack of functionality in a particular package: "It's open source. Change it yourself!" This is very often easier said than done. Furthermore, what these people are actually telling you is to change *your copy* of the source code for your own purposes, not to change the *master copy* of the source code. Getting your changes accepted into the master source base often depends more on the quality of your relationship with the current project maintainers than it does on the quality of your coding skills. I'm not complaining, mind you. I'm merely stating a fact that should not be overlooked when one is considering making changes to an existing open source software package.

So why not ship Libtool as part of Automake, rather than as a separate package? Because Libtool can quite effectively be used independently of Automake. If you wish to try Libtool by itself, then please refer to the GNU Libtool manual for more information. The opening chapters in that manual describe the use of the `libtool` script as a stand-alone product. It's really as simple as modifying your makefile commands such that the compiler, linker and librarian are called using the libtool script, and then modifying some of your command line parameters, as required by Libtool.

Public include directories

Earlier in this book, I made the statement that a project sub-directory named `include` should only contain public header files--those that expose a public interface in your project. I'm now going to add just such a header file to the Jupiter project: so, I'll create an `include` directory. I'll add this directory at the top-level of the project directory structure.

If I had multiple shared libraries, I'd have a choice to make: do I create separate `include` directories for each library in the library source directory, or do I add a single top-level `include` directory? I usually use the following rule of thumb to determine the answer to this question: if the libraries are designed to work together as a group, and if consuming applications generally use the libraries as a group, then I use a single top-level `include` directory. If, on the other hand, the libraries can be effectively used independently, and if they offer fairly autonomous sets of functionality, then I provide individual include directories in my project's library subdirectories.

In the end, it really doesn't matter much, because the header files for these libraries will be installed in entirely different directory structures than those in which they exist within your project. In fact, make sure you don't inadvertently use the same file name for headers in two different libraries in your project, or you'll probably have problems installing these files. They generally end up all together in the "`$(prefix)/include`" directory, although this default can be overridden with the `pkginclude` prefix.

I'll also add a directory for the new Jupiter shared library, called `libjupiter`. These changes require adding references to these new directories to the top-level `Makefile.am` file's `SUBDIRS` variable, and then adding corresponding makefile references to the `AC_CONFIG_FILES` macro in the `configure.ac` script:

```
$ mkdir include

$ mkdir libjup

$ echo "SUBDIRS = common include libjup src" \

   > Makefile.am

$ echo "include_HEADERS = libjupiter.h" \

   > include/Makefile.am

$ vi configure.ac

...

AC_PREREQ([2.61])

AC_INIT([Jupiter], [1.0], [bugs@jupiter.org])

AM_INIT_AUTOMAKE

LT_PREREQ([2.2])

LT_INIT([dlopen])

...

AC_CONFIG_FILES([Makefile

                 common/Makefile

                 include/Makefile
```

```
                   libjup/Makefile
                   src/Makefile])
...
```

The `include` directory's `Makefile.am` file is trivial, containing only a single line, wherein the public header file, `libjupiter.h` is referred to in an Automake `HEADERS` primary. Note that I'm using the `include` prefix on this primary. You'll recall that the `include` prefix indicates that files specified in this primary are destined to be installed in the `$(includedir)` directory (eg., `/usr/local/include`). The `HEADERS` primary is much like the `DATA` primary, in that it specifies a set of files that are to be treated simply as data to be installed without modification or pre-processing. The only really tangible difference is that the `HEADERS` primary restricts the possible installation locations to those that make sense for header files.

The `libjup/Makefile.am` file is a bit more complex, containing four lines, as opposed to the usual one or two lines:

libjup/Makefile.am

```
lib_LTLIBRARIES = libjupiter.la
libjupiter_la_SOURCES = jup_print.c
libjupiter_la_LIBADD = ../common/libjupcommon.a
libjupiter_la_CPPFLAGS = -I$(top_srcdir)/include \
 -I$(top_srcdir)/common
```

Let me analyze this file line by line. The first line is the primary one, and contains the usual prefix for libraries. The `lib` prefix indicates that the referenced products are to be installed in the `$(libdir)` directory. I might also have used the `pkglib` prefix to indicate that I wanted my libraries installed into the `$(prefix)/lib/jupiter` directory. Here, I'm using the `LTLIBRARIES` primary, rather than the older `LIBRARIES` primary. The use of this primary tells Automake to generate rules that use the `libtool` script, rather than calling the compiler and librarian (`ar`) directly to generate the products.

The second line lists the sources that are to be used for the first (and only) product. The third line indicates a set of linker options for this product. In this case, I'm specifying that the `libjupcommon.a` static library should be linked into (become part of) the `libjupiter.so` shared library.

There's an important concept regarding the `*_LIBADD` variable that you should strive to understand completely: Libraries that are consumed within, and yet built as part of the same project, should be referenced internally, using relative paths within the *build* directory hierarchy. Libraries that are external to a project generally need not be referenced explicitly at all, as the `$(LIBS)` variable should already contain the appropriate "`-L`" and "`-l`" options for those libraries. These options come from attempts made by the `configure` script to locate these libraries, using the appropriate `AC_CHECK_LIBS`, or `AC_SEARCH_LIBS` macros.

The fourth line indicates a set of C preprocessor flags that are to be used on the compiler command line for locating the associated shared library header files. These options indicate, of course, that the top-level `include` and `common` directories should be searched by the pre-processor for header files referenced in the source code. In fact, here's the new source file, `jup_print.c`:

`libjup/jup_print.c`

```
#include <libjupiter.h>

#include <jupcommon.h>



int jupiter_print(char * name)

{

   print_routine(name);

}
```

I need to include the shared library header file for access to the `jupiter_print` function's public prototype. This leads us to another general software engineering principle. I've heard it called by many names, but the one I tend to use the most is "The DRY Principle", which is an acronym that stands for Don't Repeat Yourself. C function prototypes are very useful, because when used correctly, they enforce the fact that the public's view of a function is identical to the package maintainer's view. So often, I've seen source code for a function where the source file doesn't include the header containing the public prototype for the function. It's easy to make a small change in the function or prototype, and then not duplicate it in the other location--unless you've included the public header file within the source file containing the function. Then, the compiler catches all such mistakes.

I need the static library header file because I call its function from within my public library function. Note also that I placed the public header file first--there's a good reason for this. Here is another general principle: by placing the public header file first in the source file, I can allow the compiler to check that the use of this header file doesn't depend on any other files in the project.

If the public header file has a hidden dependency on some construct (a typedef, structure or pre-processor definition) defined in internal headers like `jupcommon.h`, and if I include the public header file after `jupcommon.h`, then the dependency would be hidden by the fact that the required construct is already available in the translation unit when the compiler begins to process the public header file.

Next, I'll modify the `jupiter` application's `main` function so that it calls into the shared library instead of calling into the common static library:

`src/main.c`

```
#include <libjupiter.h>



int main(int argc, char * argv[])

{
```

```
    jupiter_print(argv[0]);

    return 0;

}
```

Here, I've changed the print function from `print_routine`, found in the static library, to `jupiter_print`, as provided by the new shared library. I've also changed the header file included at the top from `libjupcommon.h` to `libjupiter.h`.

My choices of names for the public function and header file were arbitrary, but based on a desire to provide a clean, rational and informational public interface. The name `libjupiter.h` very clearly indicates that this header file provides the public interface for the `libjupiter.so` shared library. I try to name library interface functions in such a way that they are clearly part of an interface. How you choose to name your public interface members--files, functions, structures, typedefs, pre-processor definitions, global data, etc--is up to you, but you should consider using a similar philosophy. Remember, the goal is to provide a great end-user experience.

Finally, the `src/Makefile.am` file must also be modified to use my new shared library, rather than the `libjupcommon.a` static library:

`src/Makefile.am`

```
bin_PROGRAMS = jupiter

jupiter_SOURCES = main.c

jupiter_CPPFLAGS = -I$(top_srcdir)/include

jupiter_LDADD = ../libjup/libjupiter.la

...
```

In this file, I've changed the `jupiter_CPPFLAGS` variable so that it now refers to the new `include` directory, rather than the `common` directory. I've also changed the `jupiter_LDADD` variable so that it refers to the new Libtool shared library object, rather than the `libjupcommon.a` static library. All else remains the same. Note that these changes are both obvious and simple. The syntax for referring to a Libtool library is identical to that referring to an older static library. Only the library extension is different. The Libtool library extension, `.la` stands for "libtool archive".

Take a step back for a moment: Do I actually need to make this change? No, of course not. The `jupiter` application will continue to work just fine the way it was originally set up--linking the code for the static library's `print_routine` directly into the application works equally well to calling the new shared library routine (which ultimately contains the same code). There is slightly more overhead in calling a shared library routine because of the extra level of indirection when calling though a jump table.

In a real project, you might actually leave it the way it was. Why? Because both public entry points, `main` and `jupiter_print` call exactly the same function (`print_routine`) in `libjupcommon.a`, so the functionality is identical. Why add the (slight) overhead of a call through the public interface? Well, you can take advantage of shared code. By using the shared library function, you're not duplicating code--either on disk, or in memory. Again, the DRY principle at work.

In this situation, you might now consider simply moving the code from the static library into the shared library, thereby removing the need for the static library entirely. Again, I'm going to beg your indulgence with my contrived example. In a more complex project, I might very well have a need for this sort of configuration, as such common code is often gathered together into static convenience libraries. Often, only a portion of this code is reused in shared libraries. I'm going to leave it the way it is for the sake of its educational value.

Reconfigure and build

Let me summarize where the project stands at this point. Since I've added a major new component to my project build system (Libtool), I'll add the `-i` option to the `autoreconf` command, just in case new files need to be installed:

```
$ autoreconf -i

$ ./configure

...

checking for ld used by gcc...

checking if the linker ... is GNU ld... yes

checking for BSD- or MS-compatible name lister...

checking the name lister ... interface...

checking whether ln -s works... yes

checking the maximum length of command line...

checking whether the shell understands some XSI...

checking whether the shell understands "+="...

checking for ...ld option to reload object files...

checking how to recognize dependent libraries...

checking for ar... ar

checking for strip... strip

checking for ranlib... ranlib

checking command to parse ...nm -B output...

...

checking for dlfcn.h... yes

checking for objdir... .libs
```

```
checking if gcc supports -fno-rtti...

checking for gcc option to produce PIC... -fPIC

checking if gcc PIC flag -fPIC -DPIC works...

checking if gcc static flag -static works...

checking if gcc supports -c -o file.o... yes

checking if gcc supports -c -o file.o... yes

checking whether ... linker ... supports shared...

checking whether -lc should be explicitly linked...

checking dynamic linker characteristics...

checking how to hardcode library paths...

checking whether stripping libraries is possible...

checking if libtool supports shared libraries...

checking whether to build shared libraries...

checking whether to build static libraries...

...

$
```

The first noteworthy item here is that Libtool adds *significant* overhead to the configuration process. I've only shown the output lines here that are *new* since I added Libtool. All I've added to the `configure.ac` script is the reference to the `LT_INIT` macro, and I've nearly doubled my `configure` script output. This should give you some idea of the number of system characteristics that must be examined to create portable shared libraries. Libtool does a lot of the work for you.

NOTE: In the following output examples, I've wrapped long output lines to fit publication formatting, and I've added blank lines between output lines for readability. I've also removed some unnecessary text, such as long directory names--both to increase readability and to shorten line lengths.

```
$ make

...

Making all in libjup

make[2]: Entering directory `.../libjup'
```

```
/bin/sh ../libtool --tag=CC   --mode=compile gcc

  -DHAVE_CONFIG_H -I. -I../../libjup -I..

  -I../../include -I../../common   -g -O2

  -MT libjupiter_la-jup_print.lo -MD -MP -MF

  .deps/libjupiter_la-jup_print.Tpo -c

  -o libjupiter_la-jup_print.lo

  `test -f 'jup_print.c'

    || echo '../../libjup/'`jup_print.c


libtool: compile:  gcc -DHAVE_CONFIG_H -I.

  -I../../libjup -I.. -I../../include

  -I../../common -g -O2 -MT

  libjupiter_la-jup_print.lo -MD -MP -MF

  .deps/libjupiter_la-jup_print.Tpo -c

  ../../libjup/jup_print.c  -fPIC -DPIC

  -o .libs/libjupiter_la-jup_print.o


libtool: compile:  gcc -DHAVE_CONFIG_H -I.

  -I../../libjup -I.. -I../../include

  -I../../common -g -O2 -MT

  libjupiter_la-jup_print.lo -MD -MP -MF

  .deps/libjupiter_la-jup_print.Tpo -c

  ../../libjup/jup_print.c

  -o libjupiter_la-jup_print.o >/dev/null 2>&1
```

```
mv -f .deps/libjupiter_la-jup_print.Tpo
  .deps/libjupiter_la-jup_print.Plo

/bin/sh ../libtool --tag=CC   --mode=link gcc  -g
  -O2 ../common/libjupcommon.a  -o libjupiter.la
  -rpath /usr/local/lib libjupiter_la-jup_print.lo
  -lpthread

*** Warning: Linking ... libjupiter.la against the
*** static library libjupcommon.a is not portable!

libtool: link: gcc -shared
  .libs/libjupiter_la-jup_print.o
  ../common/libjupcommon.a -lpthread
  -Wl,-soname -Wl,libjupiter.so.0
  -o .libs/libjupiter.so.0.0.0

.../ld: ../common/libjupcommon.a(print.o):
  relocation R_X86_64_32 against `a local symbol'
  can not be used when making a shared object;
  recompile with -fPIC

../common/libjupcommon.a: could not read symbols:
  Bad value

collect2: ld returned 1 exit status
```

```
make[2]: *** [libjupiter.la] Error 1

...
```

That wasn't a very pleasant experience! It appears that I have some errors to fix. I'll take them one at a time, from top to bottom.

The first point of interest is that the `libtool` script is being called with a `--mode=compile` option, which causes `libtool` to act as a wrapper script around a somewhat modified version of a standard `gcc` command line. You can see the effects of this statement in the next two compiler command lines. *Two compiler commands?* That's right. It appears that `libtool` is causing the compile operation to occur twice.

A careful examination of the differences between these two command lines shows that the first compiler command is using two additional flags: "`-fPIC`" and "`-DPIC`". The first line also appears to be directing the output file to a "`.libs`" subdirectory, whereas, the second line is saving it in the current directory. Finally, both the `STDOUT` and `STDERR` output is redirected to `/dev/null` in the second line.

This double-compile "feature" has caused a fair amount of anxiety on the Libtool mailing list over the years. Mostly, this is due to a lack of understanding of what it is that Libtool is trying to do, and why it's necessary. Using various `configure` script command line options provided by Libtool, you can force a single compilation, but doing so brings with it a certain loss of functionality, which I'll explain here shortly.

The next line renames the dependency file from `*.Tpo` to `*.Plo`. Dependency files contain `make` rules that declare dependencies between source files and referenced header files. These are generated by the C preprocessor when the `-MT` compiler option is used. (And what better tool to know about such references than the one that actually processes them!) They're then included in makefiles so that the `make` utility can properly recompile a source file, if one or more of its include dependencies have been modified since the last build. This is not really germane to an examination of Libtool, so I'll not go into any more detail here, but check the GNU Make manual for more information. The point is that one Libtool command may (and often does) execute a group of shell commands.

The next line is another call to the `libtool` script, this time using the `--mode=link` option. This option generates a call to execute the compiler in "link" mode, passing all of the libraries and linker options specified in the `Makefile.am` file.

And finally, here is first problem--a portablity warning about linking a shared library against a static library. Specifically, this warning is about linking a *Libtool* shared library against a *non-Libtool* static library. You'll soon begin to see why this might be a problem. Notice also that this is not an error. Were it not for additional errors we'll encounter later, this library would be built in spite of this warning.

After the portability warning, `libtool` attempts to link the requested objects together into a shared library named "`libjupiter.so.0.0.0`". But here the script runs into the real problem--a linker error indicating that somewhere from within `libjupcommon.a`--and more specifically within `print.o`--an Intel object relocation cannot be performed because the original source file (`print.c`) was apparently not compiled correctly. The linker is kind enough to tell me exactly what I need to do to fix the problem. It indicates that I need to compile the source code using a "`-fPIC`" compiler option.

Now, if you were to encounter this error and didn't know anything about the "`-fPIC`" option, then you'd be wise at this point to open the man page for `gcc` and study it, before willy-nilly inserting compiler or linker options until the warning or error disappears, as many inexperienced programmers are wont to do. Software engineers should understand the meaning and nuances of *every* command line option used by the tools in their projects' build systems. Why? Because otherwise they don't really know what they have when their build completes. It may work the way it should--but if it does, it's simply by luck, rather than by design. Good engineers know their tools, and the best way to learn is to study error messages and their fixes until the problem is well-understood, before moving on.

So what is "PIC" code?

When operating systems create new process address spaces, they always load the executable images at the same memory address. This magic address is system-specific. Compilers and linkers know this, and they know *what* that address is on a given system. Therefore, when they generate internal references to function calls, for example, they can generate those references as *absolute* addresses. If you were somehow able to load the executable at a different location in memory, it would simply not work properly, because the absolute addresses within the code would be incorrect. At the very least, the program would crash when the it jumped to the wrong location during a function call.
Consider Figure 1 below for a moment. Given a system whose magic executable load address is 0x10000000, this diagram depicts two process address spaces within that system. In the process on the left, an executable image is loaded correctly at address 0x10000000. At some point in the code a "jmp" instruction tells the processor to transfer control to the absolute address 0x10001000, where it continues executing instructions in another area of the program. In the process on the right, the program is loaded incorrectly at address 0x20000000. When that same branch instruction is encountered, the processor jumps to address 0x10001000, because that address is hard-coded into the program image. This, of course, fails--often spectacularly by crashing, but sometimes with more subtle and dastardly ramifications.

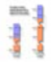
Figure 1: Absolute addressing in executable images.

That's how things work for program images. However, when a *shared library* is built for certain types of hardware (Intel x86 and x86_64 included), the address at which the library will be loaded within a process address space cannot be known by either the compiler or the linker beforehand. This is because many libraries may be loaded into any given process, and the order in which they are loaded depends on how the *executable* is built, not the library. Furthermore, who's to say which library owns location "A", and which one owns location "B"? The fact is, libraries may be loaded *anywhere* into a process where there is space for it at the time it's loaded. Only the operating system loader knows where it will finally reside--and then only just before it's actually loaded.
As a result, shared libraries can only be built from a special class of object file called "PIC" objects. PIC is an acronym which stands for "Position-Independent Code", and implies that references in the object code are not absolute, but *relative*. When the "`-fPIC`" option is used on the compiler command line, the compiler will use somewhat less efficient relative addressing in branching instructions. Such position-independent code may be loaded anywhere.
The diagram in Figure 2 below graphically depicts the concept of relative addressing, as used when generating PIC objects. When using relative addressing, regardless of where the image is loaded, addresses work correctly because they're always encoded relative to the current instruction pointer. In Figure 2, the diagrams indicate a shared library loaded at the same addresses, 0x10000000 and 0x20000000. In both cases, the DOLLAR SIGN (`$`) used in the `JMP` instruction represents the current instruction pointer (IP), so "`$ + 0xC74`" tells the processor that it should jump to the instruction starting 0xC74 bytes ahead of the current instruction pointer position.


Figure 2: Relative addressing in shared library images.

There are various nuances to generating and using position-independent code, and you should become familiar with them all before using them, so that you can choose the option that is most appropriate for your situation. For example, the GNU C compiler also supports a "`-fpic`" option (lowercase), which uses a slightly quicker, but more limited mechanism to accomplish relocatable object code. Wikipedia has a very informative page on position-independent code (although I find its treatment of Windows DLLs to be somewhat less than accurate).

Fixing the jupiter "PIC" problem

From what you now understand, one way to fix my linker error is to add the "`-fPIC`" option to the compiler command line for the source files that comprise the `libjupcommon.a` static library. Try that:

`common/Makefile.am`

```
noinst_LIBRARIES = libjupcommon.a

libjupcommon_a_SOURCES = jupcommon.h print.c

libjupcommon_a_CFLAGS = -fPIC
```

And now I'll try the build again:

```
$ autoreconf

$ make

...

gcc -DHAVE_CONFIG_H -I. -I../../common -I.. -fPIC

  -g -O2 -MT libjupcommon_a-print.o -MD -MP -MF

  .deps/libjupcommon_a-print.Tpo -c

  -o libjupcommon_a-print.o `test -f 'print.c' ||

    echo '../../common/'`print.c

...

/bin/sh ../libtool --tag=CC --mode=link gcc  -g

  -O2 ../common/libjupcommon.a -o libjupiter.la

  -rpath /usr/local/lib libjupiter_la-jup_print.lo

  -lpthread


*** Warning: Linking ... libjupiter.la against the

*** static library libjupcommon.a is not portable!
```

```
libtool: link: gcc -shared
  .libs/libjupiter_la-jup_print.o
  ../common/libjupcommon.a -lpthread -Wl,-soname
  -Wl,libjupiter.so.0 -o .libs/libjupiter.so.0.0.0

libtool: link: (cd .libs && rm -f libjupiter.so.0
  && ln -s libjupiter.so.0.0.0 libjupiter.so.0)

libtool: link: (cd .libs && rm -f libjupiter.so
  && ln -s libjupiter.so.0.0.0 libjupiter.so)

libtool: link: ar cru .libs/libjupiter.a
  ../common/libjupcommon.a
  libjupiter_la-jup_print.o

libtool: link: ranlib .libs/libjupiter.a

libtool: link: (cd .libs && rm -f libjupiter.la
  && ln -s ../libjupiter.la libjupiter.la)
...
```

I now have a shared library, built properly with position-independent code, as per system requirements. However, I still have that strange warning about the portability of linking a Libtool library against a static library. The problem here is not in what I'm doing, but rather in the *way* in which I'm doing it. You see, the concept of PIC does not apply to all hardware architectures. Some CPUs don't support any form of absolute addressing in their instruction sets. As a result, native compilers for these platforms don't support a `-fPIC` option--it has no meaning for them.

If I tried (for example) to compile my code on an IBM RS/6000 system using the native IBM compiler, it would "hiccup" when it came to the `-fPIC` option because it doesn't make sense to support such an option on a system where *all* code is automatically generated as position-independent code. One way I could get around this problem would be to make the `-fPIC` option conditional in my `Makefile.am` file, based on the type of the target system, and the tools I'm using. But that's exactly the sort of problem that Libtool was designed to address! I'd have to account for all of the different Libtool target system types and tool sets in order to handle the entire set of conditions that Libtool already handles.

The way around this portability problem then is to let Libtool generate my static library as well. Libtool makes a distinction between static libraries that are installed as part of a developer's kit, and static libraries used only internally within a project. It calls such internal static libraries "convenience" libraries, and whether or not a convenience library is generated depends on the prefix used with the `LTLIBRARIES` primary. If the `noinst` prefix is used, then Libtool assumes that I want a convenience library because there's no point in generating a shared library that will never be installed. Thus, convenience libraries are always generated as static archives.

The reason for distinguishing between convenience libraries and other forms of static library is that convenience libraries are always built, whereas non-convenience static libraries are only built if the `--enable-static` option is specified on the `configure` command line (or conversely, if the `--disable-static` option is *not* specified).

Customizing Libtool with `LT_INIT` options

Default values for enabling or disabling static and shared libraries can be specified in the argument list passed into the `LT_INIT` macro in the `configure.ac` script. Have a quick look at the `LT_INIT` macrom which may be used with or without arguments. `LT_INIT` accepts a single argument, which is a white-space separated list of key words. The following key words are valid:

- `dlopen` -- Enable checking for `dlopen` support. This option should be used if the package makes use of the `-dlopen` and `-dlpreopen` Libtool flags, otherwise Libtool will assume that the system does not support dl-opening. This option is actually assumed by default.
- `disable-fast-install` -- Change the default behavior for `LT_INIT` to disable optimization for fast installation. The user may still override this default, depending on platform support, by specifying `--enable-fast-install` to `configure`.
- `shared` -- Change the default behavior for `LT_INIT` to enable shared libraries. This is the default on all systems where Libtool knows how to create shared libraries. The user may still override this default by specifying `--disable-shared` to `configure`.
- `disable-shared` -- Change the default behavior for `LT_INIT` to disable shared libraries. The user may still override this default by specifying `--enable-shared` to `configure`.
- `static` -- Change the default behavior for `LT_INIT` to enable static libraries. This is the default on all systems where shared libraries have been disabled for some reason, and on most systems where shared libraries have been enabled. If shared libraries are enabled, the user may still override this default by specifying `--disable-static` to `configure`.
- `disable-static` -- Change the default behavior for `LT_INIT` to disable static libraries. The user may still override this default by specifying `--enable-static` to `configure`.
- `pic-only` -- Change the default behavior for `libtool` to try to use only PIC objects. The user may still override this default by specifying `--without-pic` to `configure`.
- `no-pic` -- Change the default behavior of `libtool` to try to use only non-PIC objects. The user may still override this default by specifying `--with-pic` to `configure`.

NOTE: I've omitted the description for the `win32-dll` option, because it doesn't apply to this book.

Now, back to the Jupiter project. The conversion from an older static library to a new Libtool convenience library is simple enough--all I have to do is add `LT` to the primary name and remove the `-fPIC` option and the associated variable, as there were no other options being used in that variable. Note also that I've changed the library extension from `.a` to `.la`:

`common/Makefile.am`

```
noinst_LTLIBRARIES = libjupcommon.la

libjupcommon_la_SOURCES = jupcommon.h print.c
```

`libjup/Makefile.am`

```
...

libjupiter_la_LIBADD = ../common/libjupcommon.la

...
```

Now when I try to build, here's what I get:

```
$ autoreconf

$ ./configure

...

$ make

...

/bin/sh ../libtool --tag=CC --mode=compile gcc

  -DHAVE_CONFIG_H -I. -I../../common -I..

  -g -O2 -MT print.lo -MD -MP -MF .deps/print.Tpo

  -c -o print.lo ../../common/print.c


libtool: compile: gcc -DHAVE_CONFIG_H -I.

  -I../../common -I.. -g -O2 -MT print.lo -MD -MP

  -MF .deps/print.Tpo -c ../../common/print.c

  -fPIC -DPIC -o .libs/print.o

...
```

```
/bin/sh ../libtool --tag=CC --mode=link gcc -g -O2
  -o libjupcommon.la print.lo -lpthread

libtool: link: ar cru .libs/libjupcommon.a
  .libs/print.o
...
/bin/sh ../libtool --tag=CC --mode=link gcc -g -O2
  ../common/libjupcommon.la -o libjupiter.la
  -rpath /usr/local/lib libjupiter_la-jup_print.lo
  -lpthread

libtool: link: gcc -shared
  .libs/libjupiter_la-jup_print.o
  -Wl,--whole-archive
  ../common/.libs/libjupcommon.a
  -Wl,--no-whole-archive -lpthread -Wl,-soname
  -Wl,libjupiter.so.0 -o .libs/libjupiter.so.0.0.0
...
```

You can see that the `common` library is now built as a static convenience library because the `ar` utility is used to build `libjupcommon.a`. Libtool also seems to be building files with new and different extensions. A closer look will discover extensions such as `.lo` and `.la`. If you take a closer look at these files, you'll find that they're actually descriptive text files containing object and library meta data. Take a look at the `common/libjupcommon.la` file:

common/libjupcommon.la

```
# libjupcommon.la - a libtool library file
# Generated by ltmain.sh (GNU libtool) 2.2
#
# Please DO NOT delete this file!
```

```
# It is necessary for linking the library.



# The name that we can dlopen(3).

dlname=''



# Names of this library.

library_names=''



# The name of the static archive.

old_library='libjupcommon.a'



# Linker flags that can not go in dependency_libs.

inherited_linker_flags=''



# Libraries that this one depends upon.

dependency_libs=' -lpthread'

...
```

The various fields in these files help the linker--or rather the `libtool` wrapper script--to determine certain options that would otherwise have to be remembered by the developer, and then passed on the command line to the linker. For instance, the library's shared and static names are remembered here, as well as any other library dependencies required by these libraries. In this library, for example, I can see that `libjupcommon.a` depends on the `pthread` library. But, using Libtool, I don't have to pass a `-lpthread` option on the libtool command line because `libtool` can detect in this meta data file that the linker will need this, so it passes the option for me.

Making these files human-readable was a minor stroke of genius, as they can tell me a lot about my Libtool libraries, at a glance. These files are designed to be installed with their associated binaries, and in fact, the `make install` rules generated by Automake for Libtool libraries do just this.

The Libtool library versioning scheme

If you've spent any time at all working at the Linux command prompt, then you'll certainly recognize this series of executable and link names.

NOTE: There's nothing special about `libz`--I am merely using this library as a common example:

```
$ ls -dal /lib/libz*

... /lib/libz.so.1 -> libz.so.1.2.3

... /lib/libz.so.1.2 -> libz.so.1.2.3

... /lib/libz.so.1.2.3
```

If you've ever wondered what this means, then read on. Libtool provides a versioning scheme for shared libraries that has become prevalent in the Linux world. Other operating systems use different versioning schemes for shared libraries, but the one defined by Libtool has become so popular that people often associate it with Linux, rather than with Libtool. This is not entirely an unfair assessment because the Linux loader honors this scheme to a certain degree. But to be completely fair, it's Libtool that should be given the credit for this versioning scheme.

One interesting aspect of this scheme is that, if not understood properly, people can easily mis-use or abuse the system without intending to. People who don't understand this system tend to think of the numeric values as *major*, *minor* and *revision*, when in fact, these values have very specific meaning to the operating system loader, and must be updated properly for each new library version in order to keep from confusing the loader. I remember a meeting I had at work one day several years ago with my company's corporate versioning committee. This committee's job was to come up with software versioning policy for the company as a whole. They wanted us to ensure that the version numbers incorporated into our shared library names were in alignment with the corporate software versioning standard. It took me the better part of a day to convince them that a shared library version was not related to a product version in any way, nor should such a relationship be established or enforced by them or anyone else.

Here's why. The version number on a shared library is not really a library version, but rather an interface version. The interface I'm referring to here is the application programming interface (API) presented by a library to the potential user--a programmer wishing to call functions in the interface. As the GNU Libtool manual points out, a program has a single well-defined entry point (usually called `main`, in the C language). But a shared library has multiple entry points that are generally not standardized in a widely understood manner. This makes it much more difficult to determine if a particular version of a library is "interface-compatible" with another version of the same library. *NOTE: The concept of "interface" goes much deeper in shared library versioning, referring to all aspects of a shared library's connections with the outside world. These connections include files and file formats, network connections and wire data formats, IPC channels and protocols, etc. When versioning a new public release of a shared library, all aspects of the library's interactions with the world should be taken into account.*

Microsoft DLL versioning

Consider Microsoft Windows Dynamic Link Libraries (DLLs). These are shared libraries in every sense of the word. They provide a proper application programming interface. But unfortunately, Microsoft has in the past provided no integrated DLL interface versioning scheme. As a result, Windows developers have often refered to DLL versioning issues (tongue-in-cheek, I'm sure) as "DLL hell".

As a fix to this problem, on Windows systems, DLLs can be installed into the same directory as the program that uses them, and the Windows operating system loader will always attempt to use the local copy first before searching for a copy in the system path. This alleviates a part of the problem because a specific version of the library can be installed with the package that requires it.

While this is a fair solution it's not a really good solution, because one of the major benefits of shared libraries is that they can be shared--both on disk and in memory. If every application has its own copy of a different version of the library, then this benefit of shared libraries is lost--both on disk and in memory.

Since the introduction of this partial solution, Microsoft hasn't paid much attention to DLL sharing efficiency issues. The reasons for this include both a cavalier attitude regarding the cost of disk space and RAM, and a technical issue regarding the implementation of Windows dynamic link libraries. Instead of generating position-independent code, Microsoft system architects chose to link DLL's with a specific base address, and then list all absolute address references in a base table in the image header. When a DLL can't be loaded at the required base address (because of a conflict with another DLL), then the loader "rebases" the DLL by picking a new base address and changing all of the absolute addresses referred to in the base table. Whenever a DLL is rebased in this manner, it can only be shared with processes that happen to rebase the DLL to the same address. The odds of accidentally encountering such a scenario--especially among applications with many DLL components--are pretty slim.

Recently, Microsoft invented the concept of the "Side-by-Side Cache" (sometimes referred to as "SxS"), which allows developers to associate a unique identification value (a GUID, in fact) with a particular version of a DLL installed in a system location. This location is named by the DLL name and version identifier. Applications built against SxS-versioned libraries have meta data stored in their executable headers that indicate the particularly versioned DLLs that they require. If the right version is found (by newer OS loaders) in the SxS cache, then they load it. Based on policy in the meta data, they can then revert to the older scheme of looking for a local and then a global copy of the DLL. This is a vast improvement over earlier solutions--providing a very flexible versioning system.

Given the fact that DLLs use the rebasing technique, as opposed to PIC code, the side-by-side cache is still a fairly benign improvement with respect to applications that manage dozens of shared libraries. SxS is really intended for system libraries that many applications are likely to consume. These are generally "based" at different addresses, so that the odds of clashing (and thus rebasing) are decreased. Regardless, the entire based approach to shared libraries has the major drawback that the program address space may become fairly fragmented, as randomly chosen base addresses are honored throughout a 32-bit address space by the system loader. 64-bit addressing helps tremendously in this area, so you may find the side-by-side cache to be much more useful on 64-bit Windows systems.

Linux and other Unix-like systems that support shared libraries manage interface versions using the Libtool versioning scheme. In this scheme, shared libraries are said to support a range of interface versions, each identified by a unique integer value. If any aspect of an interface changes in any way between public releases, then it can no longer be considered the same interface. It becomes a new interface, identified by a new integer interface value. To make the interface versioning process comprehensible to the human mind, each public release of a library wherein the interface has changed simply acquires the next consecutive interface version number. Thus, a given shared library may support versions 2-5 of an interface.

Libtool shared libraries follow a naming convention that encodes the interface range supported by a particular shared library. A shared library named `libname.so.0.0.0` contains the library interface version number, `0.0.0`. these three values are respectively called the library interface `current`, `revision` and `age` values.

The `current` value represents the current interface version number. This is the value that changes each time a new interface version must be declared, because the interface has changed in any way since the last public release of the library. The first interface in a library is given a version number of "0", by popular convention.

Consider a shared library wherein the developer has added a new function to the set of functions exposed by this library since the last public release. The interface can't be considered the same in this new version as it was in the previous version because there's one additional function. Thus, it's `current` number must be increased from "0" to "1".

The `age` value represents the number of back-versions supported by the shared library. In mathematical terms, the library is said to support the interface range, `current` - `age` through `current`. In the example I just gave, a new function was added to the library, so the interface presented in this version of the library is not the same as that presented in the previous version. However, the previous version is still fully supported because the previous interface is a proper subset of the current interface. Thus, this library could conceivably be named "`libname.so.1.0.1`", where the range of supported interfaces is 1 - 1 (or 0) through 1, inclusive.

The `revision` value merely represents a serial revision of the current interface. That is, if no changes are made to a shared library's interface between releases--perhaps an internal function was optimized--then the library name should change in some manner, but both the `current` and `age` values would be the same, as the interface has not changed. The `revision` value is incremented to reflect the fact that this is a new release of the same interface. If two libraries exist on a system with the same name, and the same `current` and `age` values, then the operating system loader will always select the library with the higher `revision` value.

To simplify the release process for shared libraries, the GNU Libtool manual provides an algorithm that should be followed step-by-step for each new version of a library that is about to be publically released. I'll reproduce the algorithm verbatim here for your information:

- Start with version information of `0:0:0` for each libtool library. [This is done automatically by simply omitting the `-version` option from the list of linker flags passed to the `libtool` script.]
- Update the version information only immediately before a public release of your software. More frequent updates are unnecessary, and only guarantee that the `current` interface number gets larger faster.
- If the library source code has changed at all since the last update, then increment `revision` (`c:r:a` becomes `c:r+1:a`).
- If any interfaces [exported functions or data] have been added, removed, or changed since the last update, increment `current`, and set `revision` to 0.
- If any interfaces have been added since the last public release, then increment `age`.
- If any interfaces have been removed since the last public release, then set `age` to 0.

Keep in mind that this is an algorithm, and as such it is designed to be followed step by step, as opposed to jumping directly to the steps that appear to apply to your case. For example, if you removed any API functions from your library since the last release, you would not simply jump to the last step and set `age` to zero. Rather, you would follow all of the steps properly until you reached the last step, and *then* set `age` to zero.

In greater detail: assume that this is the second release of a library, and that the first release was named `libexample.so.0.0.0`, and that one new function was added to the API during this development cycle, and one old function was deleted. The effect on this release of the library would be as follows:

- (n/a)

- (n/a)
- `libexample.so.0.0.0` -> `libexample.so.0.1.0` (library source was changed)
- `libexample.so.0.1.0` -> `libexample.so.1.0.0` (library interface was modified)
- `libexample.so.1.0.0` -> `libexample.so.1.0.1` (one new function was added)
- `libexample.so.1.0.1` -> `libexample.so.1.0.0` (one old function was removed)

Why all the "hoop jumping"? Because, as I alluded to earlier, the versioning scheme is honored by the linker and the operating system loader. When the linker creates the library name table in an executable image header, it writes the versions of the libraries linked to the application along side of each entry in this table. When the loader searches for a matching library, it looks for the latest version of the library required by the executable. If the application was linked with version 0.0.0 of a particular library, but the user only has version 1.0.1 installed, the system will load it and use it because it's `current` and `age` values indicate that it supports the required version (0).

Note also that `libname.so.0.0.0` can coexist in the same directory as `libname.so.1.0.0` without any problem. Programs that need the earlier version (which supports *only* the later interface because of the deleted function) will properly and automatically have it loaded into their process address space, just as will programs that require the later version properly have the "1.0.0" version loaded.

One more point regarding interface versioning. Once you fully understand Libtool versioning, you'll find that even the above algorithm does not cover all possible interface modification scenarios. Consider, for example, version 0.0.0 of a shared library that you maintain. Now, assume you add a new function to the interface for the next public release. This second release is properly named version 1.0.1, because the library supports both interfaces 0 and 1. Just before the third release of the library, you realize that you didn't really need that new function after all, and so you remove it. Assume also that this is the only change made to the library interface in this release. The above algorithm would have this release named version 2.0.0. But in fact, you've merely removed the second interface, and are now presenting the original interface once again. Technically, this library should be properly named version 0.1.0, as it presents a second release of version 0 of the shared library interface.

Using `libltdl` to `dlopen` a shared library

Once again, I'm going to have to add some functionality to the Jupiter project in order to illustrate the concepts of this section. The goal here is to create a plug-in interface that the `jupiter` application can use to enhance the output.

Necessary infrastructure

Currently, `jupiter` prints "Hello, from jupiter!". (Actually, the name printed is more likely at this point to be a long ugly path containing some Libtool directory garbage and some derivation of the name "jupiter", but just pretend it prints "jupiter" for now.) I'm going to add an additional parameter to the common static library `print` routine, named "salutation". This parameter will also be a character string reference, and will contain the leading word or phrase--the salutation, as it were.

Here are the changes I have to make to the files in the `common` directory:

`common/print.c`

```
...

static void * print_it(void * data)

{

   char ** strings = (char **)data;
```

```
    printf("%s from %s!\n", strings[0], strings[1]);
    return 0;
}


int print_routine(char * salutation, char * name)
{
    char * strings[] = {salutation, name};


#if ASYNC_EXEC
    pthread_t tid;
    pthread_create(&tid, 0, print_it, strings);
    pthread_join(tid, 0);
#else
    print_it(strings);
#endif
    return 0;
}
```

`common/jupcommon.h`

```
#ifndef JUPCOMMON_H_INCLUDED
#define JUPCOMMON_H_INCLUDED


int print_routine(char * salutation, char * name);


#endif  /* JUPCOMMON_H_INCLUDED */
```

And here are the changes I need to make to the files in the `libjup` and `include` directories:

libjup/jup_print.c

```
...

int jupiter_print(char * salutation, char * name)

{

   print_routine(salutation, name);

}
```

include/libjupiter.h

```
...

int jupiter_print(char * salutation, char * name);

...
```

And finally, here are the changes I need to make to `main.c` in the `src` directory:

src/main.c

```
...

#define DEFAULT_SALUTATION "Hello"



int main(int argc, char * argv[])

{

   char * salutation = DEFAULT_SALUTATION;



   jupiter_print(salutation, argv[0]);



   return 0;

}
```

To be clear, all I've really done here is parameterize the salutation in the print routines. That way, I can indicate from `main` what salutation I'd like to use. I've set the default salutation to "Hello", so that nothing will have changed from the user's perspective. The overall effect of these changes was benign. Note also that these are all source code changes. I've made no changes to the build system.

Adding a plug-in interface

Now, I can begin to discuss adding a plug-in interface to Jupiter. I'd like to make it possible to change the salutation displayed by simply changing a plug-in module. The code and build system changes required to add this functionality will be limited here to the `src` directory, and subdirectories thereof.

First, I need to define the actual plug-in interface. I'll do this by creating a new private header file in the `src` directory, called `module.h`:

src/module.h

```
#ifndef MODULE_H_INCLUDED
#define MODULE_H_INCLUDED


#define GET_SALUTATION_SYM "get_salutation"


typedef char * get_salutation_t(void);


char * get_salutation(void);


#endif  /* MODULE_H_INCLUDED */
```

There are a number of interesting points about this header file. First, the preprocessor definition, `GET_SALUTATION_SYM`. This string represents the name of the function you need to import from the plug-in module. I like to define these in the header file, so that all of the information that needs to be reconciled co-exists in one place. In this case, the symbol name, the function type definition, and the function prototype must all be in alignment. While I could have simply allowed the caller to specify the string, defining the symbol name here allows me to change it later if I need to. As long as the caller used the definition I provided, s/he should be unaffected by a name change (of course, s/he'll have to recompile).

Another interesting point is the type definition: why should I provide one? If I don't, the user is going to have to invent one, or else use a complex type cast on the return value of the `dlsym` function. I provide it here for consistency. Finally, look at the function prototype. This isn't so much for the caller, as it is for the module itself. Modules providing this function should include this header file, so that the compiler can catch potential mis-spellings of the function name. Since all of this information must be in agreement, I simply define it all here together.

Doing it the "old-fashioned" way

For this first attempt, I'll use the `dlopen/dlsym/dlclose` interface provided by the Solaris, BSD and Linux `libdl.so` library. Then, in the next section, I'll convert this code over to the Libtool `ltdl` interface. To do this right, I need to add checks to the `configure.ac` script to look for both the `libdl` library and the `dlfcn.h` header file:

configure.ac

```
...

# Checks for header files (2).

AC_CHECK_HEADERS([stdlib.h dlfcn.h])


# Checks for libraries.

# Checks for typedefs, structures, and compiler...

# Checks for library functions.

AC_SEARCH_LIBS([dlopen], [dl])

...

echo \

"-------------------------------------------------


 ${PACKAGE_NAME} Version ${PACKAGE_VERSION}


 Prefix: '${prefix}'.

 Compiler: '${CC} ${CFLAGS} ${CPPFLAGS}'

 Libraries: '${LIBS}'

...
```

These changes consist of adding the `dlfcn.h` header file to the list of files passed to the `AC_CHECK_HEADERS` macro, and adding a check for the `dlopen` function in the `dl` library. Note here that the `AC_SEARCH_LIBS` macro searches a list of libraries for a function, so this call goes under the section entitled, "Checks for library functions.", rather than the one entitled, "Checks for libraries."

To help me see which libraries I'm actually linking against, I've also added a line to the `echo` statement at the end of the file. The "Libraries:" line displays the contents of the `LIBS` variable, which is modified by the `AC_SEARCH_LIBS` macro.

NOTE: The `LT_INIT` macro actually already checks for the existence of the `dlfcn.h` header file, but I do it here explicitly, so it's obvious to observers that I wish to use this header file myself. This is a good rule of thumb to follow, as long as it doesn't negatively *affect performance too much. In this case, I felt it was well worth the extra check. Besides that, the results of the check performed by `LT_INIT` is cached by `autom4te`, so it has little effect anyway.*

Now it's time to actually add a new module. This requires several changes, so I'll make them all here now in the following command sequence:

```
$ cd src
$ mkdir -p modules/hithere
$ vi Makefile.am
SUBDIRS = modules
...
$ echo "SUBDIRS=hithere" > modules/Makefile.am
$ cd modules/hithere
$ echo "pkglib_LTLIBRARIES = hithere.la
> hithere_la_SOURCES = hithere.c
> hithere_la_LDFLAGS = -module \
>  -avoid-version" > Makefile.am
$ vi hithere.c
#include "../../module.h"

char * get_salutation(void)
{
   return "Hi there";
}
```

Okay, look for a moment at this sequence. First, I created a `modules` directory beneath the existing `src` directory, and then a `hithere` directory beneath the new `modules` directory. The `hithere` module will provide the salutation, "Hi there" to the caller. Next, I added a `SUBDIRS` directive to the top of the `src/Makefile.am` file, indicating that the new `modules` directory should be processed by Automake. Then I created a new `Makefile.am` file in the new `hithere` directory, containing instructions on how to build the `hithere.c` source file. Finally, I went ahead and added the `hithere.c` source file, itself.

The source file includes the private `module.h` header file using a double quoted relative path. The `make VPATH` statement will handle any differences between the source and build trees with regard to this relative path. The file then defines the `get_salutation` function, which is prototyped in the `module.h` header file. It simply returns a pointer to a static string.
As long as this library is loaded, this string is available to the caller. This is important to know because the caller must know the scope of data references returned by plug-in modules, as such modules could inadvertently be unloaded before the caller is ready to stop using these references.

The last line of the `hithere/Makefile.am` file requires some explanation. Here, I'm using a `-module` option on the `hithere_la_LDFLAGS` variable. This is a Libtool option, that tells Libtool that you really do want to call your library "`hithere`", and not "`libhithere`". The GNU Libtool manual makes the statement that modules do not need to be prefixed with "lib". Quite frankly, I'm not sure who came up with this policy, but it seems fairly arbitrary to me. I suppose the reason for this is that since your own code will be loading the module, it should not have to be concerned with using the "lib" prefix. Oh well, there you have it--modules need not be prefixed with "lib".
If you don't care to use module versioning on your dynamically loadable (`dlopen`) modules, then try using the Libtool `-avoid-version` option, as I've also done here. This option causes Libtool to generate the shared library as `libname.so`, rather than `libname.so.0.0.0`, along with links for `libname.so.0` and `libname.so` pointing to this binary image.
I still need to make one more change to the `configure.ac` file to get this new module to build. I need to add these two new makefiles to the `AC_CONFIG_FILES` list.

`configure.ac`

```
...
AC_CONFIG_FILES([Makefile
                 common/Makefile
                 include/Makefile
                 libjup/Makefile
                 src/Makefile
                 src/modules/Makefile
                 src/modules/hithere/Makefile])
...
```

These changes will allow our module to be built, but I'm still not using it. To use the module, I need to modify the `src/main.c` file so that it loads the module, imports the symbol, and uses it.

`src/main.c`

```
#include <libjupiter.h>
```

```
#include "module.h"

#if HAVE_CONFIG_H
# include <config.h>
#endif

#if HAVE_DLFCN_H
# include <dlfcn.h>
#endif

#define DEFAULT_SALUTATION "Hello"

int main(int argc, char * argv[])
{
   char * salutation = DEFAULT_SALUTATION;

#if HAVE_DLFCN_H
   void * module;
   get_salutation_t * get_salutation_fp = 0;

   module = dlopen("./module.so", RTLD_NOW);
   if (module != 0)
   {
      get_salutation_fp = (get_salutation_t *)
            dlsym(module, GET_SALUTATION_SYM);
      if (get_salutation_fp != 0)
```

```
        salutation = get_salutation_fp();
    }
#endif

    jupiter_print(salutation, argv[0]);

#if HAVE_DLFCN_H
    if (module != 0)
        dlclose(module);
#endif

    return 0;
}
```

In this new version of `main.c`, I'm including the new private `module.h` header file. I've also added preprocessor directives to conditionally include `config.h`, and then `dlfcn.h`. Finally, I've added two sections of code; one before and one after the original call to `jupiter_print`. Both are conditionally compiled, based on whether or not I have access to a dynamic loader. This conditional, of course, allows our code to build and run correctly on systems that do not provide run-time dynamic linking via the `libdl` library.

The general philosophy that I use here when deciding if code should be conditionally compiled is this: if I fail in the `configure` script because a library or header file is missing, then I don't need to conditionally compile the code that uses the item checked for by `configure`. If I check for a library or header file in `configure`, but allow it to continue if it's missing, then I'd better use conditional compilation.

There are just a few more minor points to bring up regarding the use of `libdl` interface functions. First, `dlopen` accepts two parameters, a file name or path (absolute or relative), and a `flags` word, which is the bitwise composite of your choice of several flag values defined in `dlfcn.h`. If a path is used, then `dlopen` honors that path verbatim. But if a file name is used, then the library search path is searched for your module. By prefixing the name with `./`, I've told `dlopen` not to search.

But, shouldn't the file name have been "hithere.so"? Well, it's true that I built a module called "hithere.so", but I want to be able to configure which module `jupiter` uses. So I'm using the generic name, "module.so". In fact, the built module is actually located several directories down in the build tree from the `src` directory. To test this functionality, I'll need to create a link in the current directory called `module.so` that points to the module I wish to load.

```
$ ./configure && make
```

```
...

$ cd src

$ ./jupiter

Hello, from ...jupiter!

$ ln -s modules/hithere/.libs/hithere.so module.so

$ ./jupiter

Hi there, from ...jupiter!

$
```

All of this would normally be done using policy defined in some sort of configuration file in a real application, but none of this is important in this example, so I'm simply ignoring these details to simplify the code.

Check the man page for `dlopen` to learn more about the flag bits that may be specified. By this point in this chapter, you should have the background required to understand most of the descriptions you'll find there.

Converting to Libtool's `ltdl` library

As I mentioned earlier, Libtool provides a wrapper library called `ltdl` that abstracts and hides some of the portability issues surrounding the use of shared libraries across many different platforms. Most applications ignore the `ltdl` library because of the added complexity involved in using it. But there are really only a few issues to deal with. I'll enumerate them here, and then cover them in detail:

- The `ltdl` functions follow a naming convention based on the `dl` library. However, the names are different. Generally, the rule of thumb is that `dl` functions such as `dlopen` are prefixed in the `ltdl` library with `lt_`. Thus, `dlopen` is named `lt_dlopen`.
- Unlike the `dl` library, the `ltdl` library must be initialized and terminated at appropriate locations in a program.
- To make full use of `ltdl` functionality--even on platforms that don't provide shared library functionality--you need to build your consuming application (the `jupiter` program, in this case), using the `-dlopen <modulename>` option on the linker command line.
- To ensure that modules can be "opened" on non-shared library platforms, or when building static-only configurations, you need to use the `LTDL_SET_PRELOADED_SYMBOLS()` macro at an appropriate location in your program source code.
- Shared library modules designed to be `dlopened` using `ldtl` should use the `-module` option (and optionally, the `-avoid-version` option) on the linker command line (specifically, in the `*_LDFLAGS` variable).
- The `ltdl` library also provides extensive functionality beyond the `dl` library, and this can be intimidating, but all of this other functionality is optional.

Take a look specifically at what I need to do to the Jupiter project build system in order to use the `ltdl` library. First, I need to modify the `configure.ac` script to look for the `ltdl.h` header, and search for the `lt_dlopen` function. This means modifying references to `dl.h` and the `dl` library in the `AC_CHECK_HEADERS` and `AC_SEARCH_LIBS` macros:

`configure.ac`

```
...

# Checks for header files (2).

AC_CHECK_HEADERS([stdlib.h ltdl.h])



# Checks for libraries.

# Checks for typedefs, structures, and compiler...

# Checks for library functions.

AC_SEARCH_LIBS([lt_dlopen], [ltdl])

...
```

If I'm using Libtool, then why do I even need to check for `ltdl.h` and `libltdl`? Because, these are separate libraries, which must be installed on your end-user's system in order to make them available.

I'd like you to recognize that this is the first time that the Autotools have required an end-user to have an Autotools package installed on his or her machine. This is the very reason is why most people avoid the use of `ltdl` entirely. The GNU Libtool manual provides a detailed description of how to package the `ltdl` library with your project, so that it get's built and installed on the end-user's system when your package is built and installed.

In fact, this tutorial (which you'll find in Section 11.6 of that manual) is a great example of adding sub-projects into a project. Interestingly, shipping the source code for the `ltdl` library with your package is the only way to get your program to statically link with the `ltdl` library. Linking statically with `ltdl` has the added (and very ironic) side effect of not requiring the `ltdl` library to be installed on the end-user's system at all! Since it becomes part of your executable images, you no longer need it to be installed. However, there are caveats to doing this. If you happen to consume a third-party library that does link dynamically to ltdl, then you'll have a symbol conflict between the shared and static versions of the `ltdl` libraries. Given how little `ltdl` is currently used, this is an unlikely scenario these days, but all of this could change in the future, if more packages begin to use `ltdl`, one way or the other.

In any case, by searching for these installed resources on the end-user's system, and by failing configuration if they're not found, or by properly using preprocessor definitions in your source code, you can provide the same sort configuration experience with `ltdl` that I've talked about throughout this book when using other third-party resources. It's the same, really.

The next major change required is found in the source code--limited, in this case, to `src/main.c`:

`src/main.c`

```
#include <libjupiter.h>

#include "module.h"
```

```
#if HAVE_CONFIG_H
# include <config.h>
#endif

#if HAVE_LTDL_H
# include <ltdl.h>
#endif

#define DEFAULT_SALUTATION "Hello"

int main(int argc, char * argv[])
{
   char * salutation = DEFAULT_SALUTATION;

#if HAVE_LTDL_H
   int ltdl;
   lt_dlhandle module;
   get_salutation_t * get_salutation_fp = 0;

   LTDL_SET_PRELOADED_SYMBOLS();

   ltdl = lt_dlinit();
   if (ltdl == 0)
   {
      module = lt_dlopen("modules/.../hithere.la");
```

```
        if (module != 0)
        {
            get_salutation_fp = (get_salutation_t *)
                    lt_dlsym(module, GET_SALUTATION_SYM);
            if (get_salutation_fp != 0)
                salutation = get_salutation_fp();
        }
    }
#endif

    jupiter_print(salutation, argv[0]);

#if HAVE_LTDL_H
    if (ltdl == 0)
    {
        if (module != 0)
            lt_dlclose(module);
        lt_dlexit();
    }
#endif

    return 0;
}
```

The changes here are very symmetrical with respect to the original code. Mostly, items that previously referred to `dl` now refer to `ltdl` or `lt_dl`. For example, `#if HAVE_DL_H` now becomes `#if HAVE_LTDL_H`, and so forth.
One important change is the fact that the `ltdl` library must be initialized with a call to `lt_dlinit`, whereas the `dl` library need not be initialized at all. This complicates the code a little--in fact, it may appear to do so much more than it really does, just by virtue of

the fact that jupiter is so ridiculously simple. In a larger program, the complexity overhead of calling lt_dlinit and lt_dlexit are amortized over a much larger code base.

Another important detail is the addition of the LTDL_SET_PRELOADED_SYMBOLS macro. This macro is used to configure global variables required by the lt_dlopen and lt_dlsym functions on systems that don't support shared libraries. It's benign on systems where shared libraries are used.

One last detail that I should mention is that the return type of dlopen was void *, or a generic pointer, whereas the return type of lt_dlopen is actually lt_dlhandle. This abstraction is important so that ltdl can be ported to systems that have a return type not compatible with a void pointer.

When a system doesn't support shared libraries, Libtool actually links all modules that might be loaded right into the program. Thus, the jupiter program's linker command line must contain some form of reference to these modules. This is done using the -dlopen <modulename> construct, in this manner:

src/Makefile.am

```
...

jupiter_LDADD = ../libjup/libjupiter.la \

 -dlopen modules/hithere/hithere.la

...
```

Now, this begs the question: What do you do when there is a choice of module to be loaded, as in the case of the jupiter program? If Libtool links them all into a program, and they all provide a get_salutation function, then there will be a conflict of public symbols. Which one will be used? The GNU Libtool manual provides for this condition by defining a convention for symbol naming:

- All exported interface symbols should be prefixed with <modulename>_LTX_ (eg., hithere_LTX_get_salutation).
- All remaining non-static symbols should be reasonably unique. The Libtool way is to prefix them with _<modulename>_ (eg., _jupiter_internal_function).
- Modules should, of course, be named differently, even if they're built in different directories.

Although (unfortunately) it's not explicitly stated in the GNU Libtool manual, the lt_dlsym function first searches for the specified symbol as <modulename>_LTX_<symbolname>, and then, if it can't find a prefixed version of the symbol, for exactly <symbolname>.

You can see that this convention, or something like it, is necessary, but only for cases where Libtool *may* statically link such loadable modules directly into the application on systems that don't support shared libraries. Libtool's ltdl library makes it possible to have the appearance of shared libraries on platforms that don't support shared libraries, but the price you have to pay for this level of portability is pretty high. This is another reason why people avoid the use of ltdl.

To fix the hithere module's source code so that it's in conformance with this convention, I have to make the following changes:

src/modules/hithere/hithere.c

```
#define get_salutation hithere_LTX_get_salutation
```

```
#include "../../module.h"

char * get_salutation(void)
{
   return "Hi there";
}
```

While it is indeed rather odd to have a preprocessor definition above a header file inclusion statement, in this case, it makes sense. By defining the replacement for `get_salutation` above the inclusion of the `module.h` header file, I'm also able to change the prototype in the header file so that it matches the modified version of the function name. Because of the way the C preprocessor works, this substitution only affects the function prototype in `module.h`, not the quoted symbol string, or the type definition.

Checking it all out

You can test your program and modules for both static and dynamic shared library systems by using the `--disable-shared` option on the `configure` command line:

```
$ ./configure --disable-shared && make
...
$ cd src
$ ls -1p modules/hithere/.libs
hithere.a
hithere.la
hithere.lai
$ ./jupiter
Hi there, from ./jupiter!
$
$ cd ..
$ make clean
...
```

```
$ ./configure && make

$ cd src

$ ls -1p modules/hithere/.libs

hithere.a

hithere.la

hithere.lai

hithere.o

hithere.so

$ ./jupiter

Hi there, from ...jupiter!

$
```

As you can see, in both configurations, the output contains the `hithere` salutation, and yet in the `--disable-shared` version, the shared library doesn't even exist. It seems that `ltdl` is doing its job.

The Jupiter code base has become rather fragile, because I've ignored the issue of where to find shared libraries at run-time. This problem would ultimately have to be fixed in a real program. But, given that I've finished my task of showing you how to properly use the Libtool `ltdl` library, and that I've taken the "Hello, world!" concept *much* farther than anyone has a right to, I think I'll just leave the rest as an exercise.

Summary

That was a lot to assimilate. Libtool, as with the other packages in the Autotools tool chain gives you a lot of functionality and flexibility. As you've probably noticed, with this functionality and flexibility comes complexity.

Libtool can make your life easier, or more difficult, depending on how you choose to use the options and flexibility it offers you. But with this background, you can decide the degree to which you will embrace the optional features of Libtool, like the `ltdl` library, for example. The decision to use shared libraries brings with it a whole truck-load of issues. Each must be dealt with if you're interested in maximum portability. The `ltdl` library is not a solution to every problem. It solves some problems, but brings others to the surface. Suffice it to say that using `ltdl` has trade-offs.

Hopefully, by spending a little time going through the exercises in this book, you've been able to "get your head around" the Autotools enough to at least be comfortable with how they work and what they're doing for you. At this point, you should be very comfortable Autotool-izing your own projects--at least at the basic level.

Source archive

Download the attached source archive for the original sources associated with this chapter.

Chapter 6: FLAIM: an Autotools example

Mon, 2008-03-10 20:34 -- John Calcote

In this book, I've taken you on a whirlwind tour of the main features of Autoconf, Automake and Libtool. I believe I've explained them in a manner that was not only simple to digest, but also to retain--especially if you had the time and inclination to follow my lead with your own copies of the examples. I've always believed that no form of learning comes anywhere close to the learning that happens while *doing*.
This chapter has downloads!
In this chapter, I'll continue this learning-by-doing pattern by converting an existing open source project to use the GNU Autotools.

The project I've chosen is called FLAIM, which is (what else?) an acronym that stands for FLexible Adaptable Information Management. FLAIM is actually a highly scalable database management system, written entirely in C++, and built on its own thin portability layer called the FLAIM Tool Kit (FTK).

What's FLAIM?!

Some of you out there may recognize FLAIM as the database used by both Novell eDirectory and the Novell GroupWise server. Novell eDirectory currently uses this particular version of FLAIM today to manage directory information bases (DIBs) containing over a billion objects. GroupWise actually uses a much earlier spin-off of FLAIM.

Novell made the FLAIM source code available as an open source project licensed under the GNU General Public License (GPL) version 2 in 2006. The FLAIM project is hosted by Novell's forge site. As a side note, if you're interested in looking at FLAIM yourself, you'll need to set up a Novell account. This is simple to do, and costs nothing. You'll be given the opportunity to create a Novell account the first time you attempt to access the Novell forge site.

Why FLAIM?

While FLAIM is not a mainstream OSS project, it has several qualities that make it the perfect choice of project to convert to GNU Autotools in this chapter. For instance, it's currently built using a hand-coded makefile--and a beast of makefile it is, too, containing well over 2000 lines of complex make script. The FLAIM makefile contains a number of GNU-make specific constructs, and thus can only be processed using GNU make. Individual (but nearly identical) makefiles are used to build the flaim, xflaim, and flaimsql database libraries, as well as the FLAIM tool kit (ftk) and several utility and sample programs on GNU/Linux, Unix, Windows and NetWare.

The existing FLAIM build system targets several different flavors of Unix, including AIX, Solaris, and HP/UX, as well as Apple's OS X. It also targets multiple compilers on these systems. These features make FLAIM ideal for my example conversion project, because I can show you how to handle differences in operating systems and tool sets in the new `configure.ac` files.
The existing build system also contains rules for many of the standard GNU Autotools targets, such as distribution tarballs. In addition, it provides rules for building binary installation packages, as well as RPMs for systems that can build and install RPM packages. Finally, it even provides targets for building doxygen description files, which it

then uses to build source documentation. I'll spend a few paragraphs discussing how these types of targets can be added to the infrastructure provided by Automake.

The FLAIM tool kit is a portability library that can be built and consumed in its own right by third-party applications or libraries. This gives me the opportunity to demonstrate Autoconf's ability to manage separate sub-projects as optional sub-directories within a project. That is, if the FLAIM tool kit happens to already be installed on the end-user's build machine, then the installed version may be used, or optionally overridden with the local copy. On the other hand, if the FLAIM tool kit is not installed, then the local, sub-directory based copy will be used by default.

The FLAIM project also provides code to build both Java and CSharp language bindings, which allows me to delve a bit into those esoteric realms. I'll not go into great detail on building either Java or CSharp applications, but I will cover how to write a `Makefile.am` file that does.
The FLAIM project makes good use of unit tests, which are built as individual programs that run without command line parameters. Thus, I can easily show you how to add unit tests to the new FLAIM build system using Automake's trivial test framework. (Autoconf supplies a more extensive test framework called Autotest, but I'll not discuss Autotest at this time.)

The FLAIM project, and its original build system happen to use a reasonably modular directory layout, making it rather easy to convert to GNU Autotools, which simply run better in projects that follow such good design principles. As one of my goals is ultimately to submit this build system back to the project maintainers, it's nice not to have to rearrange too much of the source code. A simple directory tree diff should suffice.

Finally, I also chose FLAIM because I have some limited experience with it. Although I have been given check-in rights to the project, I'm not really a FLAIM developer, and my experience is pretty much limited to using it for simple database projects occasionally.

Why hasn't FLAIM already been converted?

There are several good reasons why FLAIM hasn't already been converted to use the GNU Autotools.

- FLAIM is still a fairly new open source project, having only been released a couple of years ago.
- FLAIM's existing build system is well-understood by the developers, and they have limited experience with the GNU Autotools.
- FLAIM's build system targets three different kinds of platform, Windows, Unix and NetWare, using only GNU makefiles. This makes it difficult to give up, because one makefile is used to build FLAIM for all target platforms.

But FLAIM's build system is *not* well understood by the open source community. Since FLAIM's release into the "wild", several people have complained about FLAIM's "nasty" makefile on the FLAIM mailing lists. The GNU makefile that FLAIM uses is more or less an unmaintainable monstrosity, from the perspective of new developers. This negative attitude has an almost viral effect on the usefulness of the entire project within the community.
These community critics are accurate in their assessment of FLAIM's build system, with respect to an open source project. The FLAIM team recognizes this and has voiced the desire to establish an Autotools build system, at least for GNU/Linux and Unix platforms. This means that duplicate build systems would have to be created for NetWare and Windows (as per my personal philosophy with respect to using Autotools on non-Unix systems). But, as they say in the shoe business, "The customer is always right!".

An initial look

Let me just start by saying that converting FLAIM from GNU makefiles to an Autotools build system is a non-trival project. It took me a couple of weeks. Much of that time was spent determining exactly what to build, and how to do it--in other words, analyzing the existing FLAIM build system. Another significant portion of my time was spent on converting aspects of the FLAIM build system that lay on the outer fringes of Autotools functionality. For example, I spent more time converting build system rules for building CSharp language bindings than I did for building the core C++ FLAIM libraries.

Working on the outer fringes of Autotools capabilities can be a frustrating experience. I'll readily admit that this is where most people get disgusted with the GNU Autotools--especially with Automake. It's my hope that this Chapter will put you ahead of most others in this area. Once you learn a few tricks, working on the outer fringe is pretty simple.

The first step in this conversion project is to analyze the existing directory structure and build system. What components are actually built, and which components depend on others? Can individual components be built, distributed and consumed independently? These types of component-level relationships are important, because they'll often determine how you want to layout your project directory structure.

The FLAIM project is actually several small projects, combined into one large umbrella project within its Subversion repository. There are three separate and distinct database products, flaim, xflaim and flaimsql. The flaim sub-project is the original FLAIM database library used by eDirectory and GroupWise. The xflaim project is a hierarchical XML database, optimized for node-based access. This version was developed for internal projects at Novell. The flaimsql project is FLAIM with integrated SQL semantics exposed through the FLAIM API. This was an experiment, which frankly isn't quite finished.

The point is that all three of these database libraries are separate and unrelated to each other; none of them depend on the others. Since they may easily be used independently of one another, they can actually be shipped as individual distributions. Each could be considered an individual project, in its own right. This, then will become one of my primary goals--to allow the FLAIM project to be easily broken up into smaller projects, which may be managed independently of one another.

The FLAIM tool kit is also an independent project. While it's tailored specifically for the FLAIM database projects, providing just the system service abstractions required for a DBMS, it depends on nothing but itself, and may easily be used as the basis for portability within another project, without dragging any unnecessary database baggage along. As you might guess, its file I/O abstraction is highly optimized.

The existing FLAIM project is laid out in its Subversion repository like this:

```
trunk
  flaim
    flaim
      sample
```

```
    src
    util
  ftk
    src
    util
  sql
    src
  xflaim
    csharp
    java
    sample
    src
    util
```

The complete tree is fairly deep and broad, and there are significant utilities, tests and other such binaries that are built by the existing FLAIM build system. At some point during the downward trek into this hierarchy, I have to simply stop and consider whether it's worth converting that additional utility or layer. If I don't, this chapter will be as long as all the others combined!

To this end, I've decided to convert:

- the libraries themselves
- the unit and library tests
- the utilities and other such high-level programs found in the various `util` directories
- the Java and CSharp language bindings.

I'll also convert the CSharp unit tests, but I won't go into the Java unit tests because (believe it or not), attempting to work within the Automake-provided Java framework is more painful than just writing the rules yourself. Since Automake provides no help for CSharp, I have to provide everything myself.

Getting started

My first true design decision was centered around how to organize this one FLAIM project into sub-projects. As it turns out, the existing layout is perfect for what I've ultimately done. I've created a master `configure.ac` file in the top-level `flaim` directory--the one just under `trunk`. This `configure.ac` file acts as a sort of Autoconf control file for each of the four lower-level projects, ftk, flaim, flaimsql and xflaim.
I've managed the database library dependencies on the FLAIM tool kit (ftk) by treating it as a pure external dependency, defined by make variables `FTKINC` and `FTKLIB`. In this

way, I've conditionally defined these variables to point to one of a couple of different sources, including installed libraries, or even user-specified configure options.

Adding the `configure.ac` scripts

The directory structure under the Autotools build system won't change much. In the following directory layout, I've indicated where I've placed individual `configure.ac` files. You'll recall that each `configure.ac` file represents a separate and individual project, which may be packaged and distributed independently.

```
trunk
  flaim        configure.ac (master)
    flaim      configure.ac (flaim)
      sample
      src
      util
    ftk        configure.ac (ftk)
      src
      util
    sql        configure.ac (flaimsql)
      src
    xflaim     configure.ac (xflaim)
      csharp
      java
      sample
      src
      util
        java
```

After these design decisions were made, the next task was to create these `configure.ac` scripts. The top-level script was trivial, so I created it by hand. The project-specific scripts were more complex, so I allowed the `autoscan` utility to do the bulk of the work for me. Right now, take a look at that top-level `configure.ac` script:

```
#                  -*- Autoconf -*-
```

```
# Process this file with autoconf to produce a c...


AC_PREREQ([2.62])
AC_INIT([flaim-projects], [1.0])
AC_CANONICAL_SYSTEM
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
LT_PREREQ([2.2])
LT_INIT([dlopen])


AC_CONFIG_MACRO_DIR([m4])
AC_CONFIG_SUBDIRS([ftk flaim sql xflaim])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

This file is short and simple, because it doesn't do much. Nevertheless, there are some new and important concepts in this file that I'd like to discuss. Since its only job is to configure several lower-level projects, I've taken some shortcuts. The project name and version number, for instance, are really rather unimportant, as this project will probably never be distributed in one large tarball. Regardless, *some* values had to be used, so I invented the name `flaim-projects`, and the version number `1.0`. These are not likely to change unless really dramatic changes take place in the project directory structure in the future.

The most important aspect of this script is the use of the `AC_CONFIG_SUBDIRS` macro. This new macro, which I haven't yet covered in this book, lists the sub-projects to be built, along with the current project. This macro is effectively the Autoconf equivalent of the Automake `SUBDIRS` variable. It allows the maintainer to set up a hierarchy of projects, in much the same way that `SUBDIRS` configures the directory hierarchy for Automake within a single project.

Because the four sub-projects actually contain all of the functionality, this `configure.ac` script acts simply as a control file, passing all specified configuration options to each of the sub-projects successively, in the order that they're specified in `AC_CONFIG_SUBDIRS`. The ordering is important, because the FLAIM tool kit project must be built first, since the other projects depend on it.

Another important new concept in this file is the use of the `AC_CANONICAL_SYSTEM` macro. This macro causes the environment variables, `$host`, `$build` and `$target` to be defined. These variables contain canonicalized CPU, operating system and manufacturer values for the host, build and target systems. This information can easily be parsed later in the `configure.ac` file in order to configure system-specific options. I'll dive more deeply into this concept in the project-specific scripts below.

Automake in the umbrella project

Automake usually requires the existence of several text files in the top-level project directory. These include the `AUTHORS`, `COPYING`, `INSTALL`, `NEWS`, `README`, and `ChangeLog` files. In the case of this umbrella project, it would be nice not to have to deal with these files, as they are rather redundant here. I could do this by not using Automake at all, but then I'd either have to create my own `Makefile.in` template for this directory, or use Automake once to generate one for me. I could then check this template into the repository as part of the project, along with the `install-sh` and `missing` scripts that are installed by `autoreconf -i`. Once I have these files in place, I could then remove the `AM_INIT_AUTOMAKE` macro from the master `configure.ac` file, and Autoconf will create the final `makefile` from the preserved template.

Another option would be to keep the `AM_INIT_AUTOMAKE` macro, and use the `foreign` option in the macro's optional parameter. The `foreign` option tells Automake that the project will not follow GNU standards, and thus Automake will not require the usual GNU project text files. This is the path I decided to take, because I might wish to alter the list of subordinate projects at some point in the future, and I don't want to have to hand-tweak the generated `Makefile.in` template.

The `AM_INIT_AUTOMAKE` parameter contains a string of white-space separated options that should be assumed by Automake. When Automake parses the `configure.ac` script, it notes these options, and enables them as if they'd been passed on the command line. I've also passed the `-Wall` and `-Werror` options, which indicate that Automake should enable all (Automake) warnings, and report them as errors. Note that these options have nothing to do with the compilation environment--only Automake processing.

Why add the Libtool macros?

You may be wondering at this point why I've included those expensive Libtool macros. The reason is more complicated than I wish it were. Even though I don't do anything with Libtool in the umbrella project, the lower level projects expect that a containing project will provide all the necessary scripts, and the `LT_INIT` macro provides the `ltmain.sh` script.

If you don't initialize Libtool in the umbrella project, then tools like `autoreconf`, which actually look in the *parent* directory to determine if the current project is itself a sub-project, will fail when it can't find scripts that *its* `configure.ac` file requires. For instance, within the ftk project's top-level directory, `autoreconf` expects to find a file called `../ltmain.sh`. Note the reference to the parent directory--`autoreconf` noticed by examining the parent directory that ftk was actually a sub-project of a larger project. Rather than install all of the auxilliary scripts multiple times, it causes sub-projects to look in their parent project's directory for them, so they can be installed once in a multi-project package.

If I don't use `LT_INIT` in the umbrella project, then I can't successfully run `autoreconf` in the sub-projects, because the `ltmain.sh` file will not have been installed in the parent project's top-level directory.

NOTE: For the rather small disk space savings it provides, I personally don't think it's worth breaking modularity in this manner just to manage this odd child-to-parent relationship.

Adding a macro sub-directory

Another new construct used in the top-level `configure.ac` script is the `AC_CONFIG_MACRO_DIR` macro. This macro indicates the name of a sub-directory in which the `aclocal` utility can find all project-local M4 macro files. These files are ultimately combined into the `aclocal.m4` file used by Autoconf. The use of this macro replaces the original single `acinclude.m4` file with a directory containing `.m4` files.

NOTE: This entire system of combining (one or more) M4 macro files into a single `aclocal.m4` file is a bit of a band-aid over a system that was never originally designed *for more than one macro file. In my opinion, it could use a major overhaul, by doing away with `aclocal` entirely, and just having Autoconf read the macro files in the specified (or defaulted) macro directory, along with other macro files found in system locations.*

I've indicated by the parameter to this macro that all of the local macro files to be added to `aclocal.m4` can be found in a sub-directory called `m4`. As a side benefit, when `autoreconf -i` is run, and then, when it subsequently executes the required Autotools with their respective "add missing" options, these tools will note the use of `AC_CONFIG_MACRO_DIR` in `configure.ac`, and add all missing required system macro files to the `m4` directory.

The actual reason for my choosing to do this is that Libtool will not add its additional macro files to the project if you haven't enabled the macro directory option in this manner. Instead, it complains loudly that *you* should add these files to `acinclude.m4` yourself. I found that none of the macros in the Libtool system macro files were required by my project, but that didn't stop it from complaining, and it may not be the case for your projects.

Since I wanted the Autotools to do the job for me, and this is a fairly complex project anyway, I decided to begin using this "macro sub-directory" feature. In point of fact, a future release of Autotools will require this form anyway, as it's considered the more modern way of adding macro files to `aclocal.m4`, as opposed to using a single user-generated `acinclude.m4` file.

The top-level `Makefile.am` file

The only other point to be covered regarding the umbrella project is the top-level `Makefile.am` file. This file contains the following code:

```
ACLOCAL_AMFLAGS = -I m4

EXTRA_DIST = libflaim.changes libxflaim.changes

SUBDIRS = ftk flaim sql xflaim

rpms srcrpm:
        for dir in $(SUBDIRS); do \
          $(MAKE) -C $$dir $@; \
        done

.PHONY: rpms srcrpm
```

The `ACLOCAL_AMFLAGS` variable is a requirement of using a macro sub-directory. According to the Automake documentation, this variable should be defined in the top-level `Makefile.am` file of any project that uses `AC_CONFIG_MACRO_DIR` in its

configure.ac file. These flags indicate to aclocal where it should look for macro files when it's executed by rules defined in Makefile.am.
I've used the EXTRA_DIST variable here to ensure that additional top-level files get distributed. This isn't critical to the umbrella project, since I don't intend to create distributions at this level, but I like to be complete.
The SUBDIRS variable is a duplicate of the information in the configure.ac file's AC_CONFIG_SUBDIRS macro.
I'll discuss the remaining code later, when I cover adding new make targets to your build system. These particular targets allow the end-user to build RPM packages for rpm-based GNU/Linux systems.

The sub-projects

Each of the sub-projects, flaim, ftk, flaimsql and xflaim, are set up just as in the Jupiter project. I'll start with the FLAIM toolkit (ftk) project. Because all of the others are dependent on it, it will have to be built first, anyway.

This configure.ac script was generated for me by autoscan. Autoscan is a bit finicky when it comes to where it will look for information. If your project doesn't contain a makefile file named exactly "Makefile", or if your project already contains an Autoconf Makefile.in template, then autoscan will not add any information about required libraries to the configure.scan output file. It has no other way of determining this information, except by looking into your old build system, and it won't do this unless conditions are just right.
As mentioned earlier, the FLAIM project did contain a rather large makefile, and frankly I was quite impressed with autoscan's ability to parse it for library information, given the complex nature of this multi-platform GNU makefile. Here's a snippet of the ftk project's configure.scan file:

```
...

AC_PREREQ(2.62)

AC_INIT(FULL-PACKAGE-NAME, VERSION,

  BUG-REPORT-ADDRESS)

AC_CONFIG_SRCDIR([util/ftktest.cpp])

AC_CONFIG_HEADERS([config.h])



# Checks for programs.

AC_PROG_CXX

AC_PROG_CC

AC_PROG_INSTALL
```

```
# Checks for libraries.

# FIXME: Replace `main' with a function in `-lc':

AC_CHECK_LIB([c], [main])

# FIXME: Replace `main' with a function in...

AC_CHECK_LIB([crypto], [main])

...

AC_CONFIG_FILES([Makefile])

AC_OUTPUT
```

I substituted real values for the place-holder values left by `autoscan` in the `AC_INIT` macro. I added calls to `AM_INIT_AUTOMAKE`, `LT_PREREQ` and `LT_INIT`. I added a call to `AC_CONFIG_MACRO_DIR` here, as well. Why not? I'd already done it in the umbrella project above, and this, after all, is the new "UL Approved" method for managing project-local macro files. I then changed the `AC_CONFIG_SRCDIR` file that `autoscan` recommended, for one that made more sense to me. And I deleted the use of the `AC_PROG_CC` macro; this project is written entirely in C++.

Next, I deleted the comments above each of the `AC_CHECK_LIB` macro calls, and then I started to replace the `main` place-holders in these macros with actual library function names. I say I *started* to do that, but I stopped because I wondered if all of those libraries were really necessary. Sometimes I've noticed, where hand-coded build systems are concerned, the author will often cut and paste sets of library names into the makefile until the program builds and runs correctly. (For some reason, this activity is especially prevalent when libraries are being built, although programs are not immune to it.) Also, since `autoscan` build this list by parsing the original makefile, I figured it probably tried to include everything that it thought might be a library.

Instead of blindly continuing this trend, I chose to simply comment out all of the calls to `AC_CHECK_LIB`, and see how far I was able to get in the build, and then add them back in one at a time, as required, in order to resolve missing symbols during the build. Unless your project consumes literally hundreds of libraries, this only takes a few extra minutes, but it can save you a lot of time later when builds are speedier than they otherwise might be. And personally, I like to be accurate in my build systems, using only those libraries that really are required. When used religiously, this ideology is also a good form of project-level documentation.

The `configure.scan` file contained 14 such calls to `AC_CHECK_LIB`. As it turned out, only three of them were actually required by the FLAIM tool kit on my 64-bit Linux system, `pthread`, `ncurses`, and `rt`. So I deleted the "cruft" and swapped out the place-holder parameters for real functions in the remaining three.

Finally, I added references to `src/Makefile` and `util/Makefile` to the `AC_CONFIG_FILES` macro, and then added the `echo` statement at the bottom, for some visual verification of my configuration status.

Note that I left all of the header file and library function checks in place, as originally specified by `autoscan`. I figure that `autoscan` is probably pretty accurate in noting the use of header files and functions in my source code. Who am I to argue?

Here's the final ftk `configure.ac` file (*slightly edited, as usual, to satisfy column width requirements*):

```
#                 -*- Autoconf -*-

# Process this file with autoconf to produce a c...


AC_PREREQ([2.62])

AC_INIT([FTK], [1.1], [flaim-users@forge.novell.com])

AM_INIT_AUTOMAKE([-Wall -Werror])

LT_PREREQ([2.2])

LT_INIT([dlopen])


AC_LANG(C++)


AC_CONFIG_MACRO_DIR([m4])

AC_CONFIG_SRCDIR([src/flaimtk.h])

AC_CONFIG_HEADERS([config.h])


# Checks for programs.

AC_PROG_CXX

AC_PROG_INSTALL


# Checks for optional programs.

AC_PROG_TRY_DOXYGEN


# Configure options: --enable-debug[=no].

AC_ARG_ENABLE([debug],

  [AS_HELP_STRING([--enable-debug],

    [enable debug code (default is no)])],
```

```
  [debug="$withval"], [debug=no])

# Configure option: --enable-openssl[=no].
AC_ARG_ENABLE([openssl],
  [AS_HELP_STRING([--enable-openssl],
    [enable the use of openssl (default is no)])],
  [openssl="$withval"], [openssl=no])

# Check for doxygen program.
if test -z "$DOXYGEN"; then
  echo "-----------------------------------------"
  echo " No Doxygen program found - continuing"
  echo " without Doxygen documentation support."
  echo "-----------------------------------------"
fi
AM_CONDITIONAL([HAVE_DOXYGEN],[test -n "$DOXYGEN"])

# Checks for libraries.
AC_CHECK_LIB([ncurses], [initscr])
AC_CHECK_LIB([pthread], [pthread_create])
AC_CHECK_LIB([rt], [aio_suspend])
if test "x$openssl" = xyes; then
  AC_DEFINE([FLM_OPENSSL], [],
    [Define to use openssl])
  AC_CHECK_LIB([ssl], [SSL_new])
  AC_CHECK_LIB([crypto], [CRYPTO_add])
```

```
  AC_CHECK_LIB([dl], [dlopen])
  AC_CHECK_LIB([z], [gzopen])
fi

# Checks for header files.
AC_HEADER_RESOLV
AC_CHECK_HEADERS([arpa/inet.h fcntl.h limits.h \
malloc.h netdb.h netinet/in.h stddef.h stdlib.h \
string.h strings.h sys/mount.h sys/param.h \
sys/socket.h sys/statfs.h sys/statvfs.h \
sys/time.h sys/vfs.h unistd.h utime.h])

# Checks for typedefs, structures, and compiler ...
AC_HEADER_STDBOOL
AC_C_INLINE
AC_TYPE_INT32_T
AC_TYPE_MODE_T
AC_TYPE_PID_T
AC_TYPE_SIZE_T
AC_CHECK_MEMBERS([struct stat.st_blksize])
AC_TYPE_UINT16_T
AC_TYPE_UINT32_T
AC_TYPE_UINT8_T

# Checks for library functions.
AC_FUNC_LSTAT_FOLLOWS_SLASHED_SYMLINK
```

```
AC_FUNC_MALLOC

AC_FUNC_MKTIME

AC_CHECK_FUNCS([atexit fdatasync ftruncate getcwd \

gethostbyaddr gethostbyname gethostname gethrtime \

gettimeofday inet_ntoa localtime_r memmove memset \

mkdir pstat_getdynamic realpath rmdir select \

socket strchr strrchr strstr])


# Configure DEBUG source code, if requested.

if test "x$debug" = xyes; then

  AC_DEFINE([FLM_DEBUG], [],

    [Define to enable FLAIM debug features])

fi


# Configure global pre-processor definitions.

AC_DEFINE([_REENTRANT], [],

  [Define for reentrant code])

AC_DEFINE([_LARGEFILE64_SOURCE], [],

  [Define for 64-bit data files])

AC_DEFINE([_LARGEFILE_SOURCE], [],

  [Define for 64-bit data files])


# Configure supported platforms' compiler and li...

case $host in

  sparc-*-solaris*)

    LDFLAGS="$LDFLAGS -R /usr/lib/lwp"
```

```
    if "x$CXX" != "xg++"; then
      if "x$debug" = xno; then
        CXXFLAGS="$CXXFLAGS -xO3"
      fi
      SUN_STUDIO=`"$CXX" -V | grep "Sun C++"`
      if "x$SUN_STUDIO" = "xSun C++"; then
        CXXFLAGS="$CXXFLAGS -errwarn=%all\
-errtags -erroff=hidef,inllargeuse,doubunder"
      fi
    fi ;;

  *-apple-darwin*)
    AC_DEFINE([OSX], [],
      [Define if building on Apple OSX.]) ;;

  *-*-aix*)
    if "x$CXX" != "xg++"; then
      CXXFLAGS="$CXXFLAGS -qthreaded -qstrict"
    fi ;;

  *-*-hpux*)
    if "x$CXX" != "xg++"; then
      # Disable "Placement operator delete
      # invocation is not yet implemented" warning
      CXXFLAGS="$CXXFLAGS +W930"
    fi ;;
```

```
esac

AC_CONFIG_FILES([Makefile
                 docs/Makefile
                 docs/doxyfile
                 obs/Makefile
                 obs/ftk.spec
                 src/Makefile
                 util/Makefile])

AC_OUTPUT

echo "
  ($PACKAGE_NAME) version $PACKAGE_VERSION
  Prefix.........: $prefix
  Debug Build....: $debug
  Using OpenSSL..: $openssl
  C++ Compiler...: $CXX $CXXFLAGS $CPPFLAGS
  Linker.........: $LD $LDFLAGS $LIBS
  Doxygen........: ${DOXYGEN:-NONE}
"
```

Note that I did *not* use the `foreign` keyword in the `AM_INIT_AUTOMAKE` macro this time. This is a real project, and I expect it will be packaged as such. Thus, the developers will (should) want these files. I used the `touch` command to create empty versions of the GNU project text files.

Another new construct near the top of the file is the `AC_LANG` macro. This macro indicates which language should be assumed when executing compilation tests within the `configure` script. I've passed "`C++`" as the parameter, so that Autoconf will generate compilation tests using the C++ compiler via the `$CXX` variable, rather than the default C code using the `$CC` macro.

Moving down a few more lines will have you staring at a macro called `AC_PROG_TRY_DOXYGEN`. Try as you might, you won't find this macro in the Autoconf documentation, because I wrote it myself. Here's the source code, which can be found in `ftk/m4/ac_prog_try_doxygen.m4` in the sample code download archive:

```
AC_DEFUN([AC_PROG_TRY_DOXYGEN],[

AC_REQUIRE([AC_EXEEXT])dnl

test -z "$DOXYGEN" &&\

 AC_CHECK_PROGS([DOXYGEN], [doxygen$EXEEXT])dnl

])
```

The macro tests first to see if the end-user has already set the `DOXYGEN` environment variable. If not, it then uses the standard `AC_CHECK_PROG` macro to locate it on the host machine, if it's installed. If `AC_CHECK_PROG` finds it, it sets the `DOXYGEN` variable to the name of the program, allowing the build system to later locate the actual executable in the system path. If it's not found, it doesn't set the `DOXYGEN` variable.

There are other more standard macros that check for specific programs. In fact, as simple as this macro is, I could have just used `AC_CHECK_PROGS` in the `configure.ac` file, instead of writing my own macro. I wanted to encapsulate the "test and check" construct:

```
test -z "$DOXYGEN" && AC_CHECK_PROGS...
```

Additionally, I knew I'd need this test in each of the four projects, so it was simpler to create a macro file that could just be copied into the individual projects' m4 directories. Besides, and probably most importantly for this chapter, it's more readable to see `AC_PROG_TRY_DOXYGEN`, than to see `test -z...`.

Why `AC_PROG_TRY_DOXYGEN` and not simply `AC_PROG_DOXYGEN`? Because traditionally, the `AC_PROG_*` macros fail the configuration process if the associated program is not found. I wanted the DOXYGEN variable to be populated if the `doxygen` program was found on the system, but be left empty otherwise. That way I could conditionally build the doxygen documentation.

In fact, if you look a bit farther down, you'll see some text that looks like this:

```
...

# Check for doxygen program.

if test -z "$DOXYGEN"; then

  echo "-----------------------------------------"

  echo " No Doxygen program found - continuing"

  echo " without Doxygen documentation support."
```

```
    echo "-----------------------------------------"

  fi

  AM_CONDITIONAL([HAVE_DOXYGEN],[test -n "$DOXYGEN"])

  ...
```

This tests whether or not my `AC_PROG_TRY_DOXYGEN` macro actually found a `doxygen` program, and acts on the results. If doxygen is not installed on the user's system, then the configure script prints out a large, hard-to-miss message stating that doxygen documentation will not be built. No big deal, really, unless the user was, in fact, counting on it. In that case, she can simply install doxygen and rebuild.

The `AM_CONDITIONAL` macro defines an automake variable called `HAVE_DOXYGEN`, which can be used in the project's `Makefile.am` files to do something conditionally, based on whether or not doxygen can successfully be called (via the `$DOXYGEN` variable). The first parameter is the Automake conditional variable to be defined, and the second parameter is the test to be run by the configure script in order to determine *how* the variable should be defined in the makefile. Just one caveat: `AM_CONDITIONAL` must not be used conditionally (eg., within a shell `if` statement) in the `configure.ac` script.

Immediately following the `DOXYGEN` `AM_CONDITIONAL` statement, you'll find the library checks. The first three are the ones that autoscan told me about that I found I actually needed after experimenting a bit. The next four are checked within an `if` statement. Additionally, a preprocessor macro is defined using the `AC_DEFINE` macro:

```
  ...

  if test "x$openssl" = xyes; then

    AC_DEFINE([FLM_OPENSSL], [],

      [Define to use openssl])

    AC_CHECK_LIB([ssl], [SSL_new])

    AC_CHECK_LIB([crypto], [CRYPTO_add])

    AC_CHECK_LIB([dl], [dlopen])

    AC_CHECK_LIB([z], [gzopen])

  fi

  ...
```

These libraries are included conditionally based on the user's use of the `--enable-openssl` command-line argument defined in a previous call to the `AC_ARG_ENABLE` macro. The `openssl` variable is defined to either `yes` or `no`, based on the default value given to `AC_ARG_ENABLE`, and the user's command-line choices.

The `AC_DEFINE` macro call ensures that the C++ preprocessor variable, `FLM_OPENSSL` is defined in the `config.h` file, and the `AC_CHECK_LIB` macro calls ensure that `-lssl`,

`-lcrypto`, `-ldl`, and `-lz` strings are added to the `$LIBS` variable. But *only* if the `openssl` macro is defined as `yes`.

The last item I'll cover here is the conditional use of the `AC_DEFINE` macro, based on the contents of the `debug` variable:

```
...

# Configure DEBUG source code, if requested.

if test "x$debug" = xyes; then

  AC_DEFINE([FLM_DEBUG], [],

    [Define to enable FLAIM debug features])

fi

...
```

This is another preprocessor definition, conditionally defined, based on the results of a command-line parameter given to `configure`. The `--enable-debug` option ultimately enables the definition of `FLM_DEBUG` within `config.h`. Both `FLM_OPENSSL` and `FLM_DEBUG` are consumed within the FLAIM project source code. Using `AC_DEFINE` in this manner allows the user to determine what sort of features are compiled into his binaries.

I'll cover the details of the platform-specific checks later in this chapter. This code is identical in all of the projects' `configure.ac` scripts, as the four original GNU makefiles contained identical such checks.

The `ftk/Makefile.am` file

Discounting the code for doxygen and rpm targets, the `ftk/Makefile.am` file is fairly trivial:

```
ACLOCAL_AMFLAGS = -I m4



EXTRA_DIST = COPYRIGHT GNUMakefile netware



SUBDIRS = src util obs



if HAVE_DOXYGEN

  SUBDIRS += docs

endif
```

```
doc_DATA = AUTHORS ChangeLog COPYING COPYRIGHT INSTALL NEWS
README



rpms srcrpm: dist

        $(MAKE) -C obs $(AM_MAKEFLAGS) $@

        rpmarch=`rpm --showrc | grep ^build\ arch | sed 's/\(.*:
\)\(.*\)/\2/'`; \

        test -z $$rpmarch || ( mv $$rpmarch/* .; rm -rf
$$rpmarch )

        -rm -rf $(distdir)



dist-hook:

        -rm -rf `find $(distdir) -name .svn`



.PHONY: srcrpm rpms
```

Here, you find the usual `ACLOCAL_AMFLAGS`, `EXTRA_DIST` and `SUBDIRS` variable definitions. But you can also see the use of an Automake conditional. The `if` statement allows us to append another directory (`docs`) to the `SUBDIRS` list, but only if you have access to the doxygen program. If you try to use such a conditional without a corresponding `AM_CONDITIONAL` in the `configure.ac` file, then Automake will complain about it.

Another new construct--at least in a top-level `Makefile.am` file--is the use of the `doc_DATA` variable. The FLAIM toolkit provides some extra documentation files in its top-level directory that I'd like to have installed. By using the `doc` prefix on the `DATA` primary in this manner, I'm telling Automake that I'd like to have these files installed as data files in the `$(docdir)` directory, which ultimately resolves to the `$(prefix)/share/doc` directory.

An interesting effect of the use of the `DATA` primary is that files mentioned in `DATA` variable are not automatically distributed, so you have to mention them in the `EXTRA_DIST` variable as well. You'll note that I did *not* have to mention the standard GNU project text files in `EXTRA_DIST`. These are always distributed automatically. However, I *did* have to mention the standard text files in the `doc_DATA` variable. This is because Automake makes no assumptions about the files that you want installed.

Once again, I'll defer a discussion of the RPM targets until later.

Automake "-hook" and "-local" rules

At this point, I'd like to discuss the use of the `dist-hook` target. Automake recognizes two types of extensions. I call these `-local` targets and `-hook` targets. Both of these types of targets represent Automake extension points. Automake recognizes and honors `-local` extensions for the following standard Automake targets:

- `all`
- `info`
- `dvi`
- `ps`
- `pdf`
- `html`
- `check`
- `install-data`
- `install-dvi`
- `install-exec`
- `install-html`
- `install-info`
- `install-pdf`
- `install-ps`
- `uninstall`
- `installdirs`
- `installcheck`
- `mostlyclean`
- `clean`
- `distclean`
- `maintainer-clean`

Adding a `-local` version of any of these to your `Makefile.am` files will cause Automake to ensure that the commands associated with these rules are executed *before* the associated standard target. Automake does this by generating the rule for the standard target such that the `-local` version is one of its dependencies (if it exists), thus the `-local` version is run before the commands for the standard target. Shortly, I'll show an example of this, using a `clean-local` target.

The `-hook` targets are a bit different in that they are executed *after* the corresponding standard target is executed. Automake does this by adding another command to the end of the standard target command list that executes `make` (via the `$(MAKE)` variable) on the same `Makefile`, with the `-hook` target as the command-line target. Thus, the `-hook` target is executed at the end of the standard target commands.

The following standard Automake targets support `-hook` versions:

- `install-data`
- `install-exec`
- `uninstall`
- `dist`
- `distcheck`

In this example, I use the `dist-hook` target to "adjust" the distribution directory before Automake create a tarball from its contents.

```
...

dist-hook:

        -rm -rf `find $(distdir) -name .svn`

...
```

The `rm` command removes extraneous files and directories that become part of the distribution directory as a result of my adding entire directories to the `EXTRA_DIST` variable. When you add a directory name to `EXTRA_DIST`, *everything* in that directory is added to the distribution--even hidden Subversion control files and directories. I certainly don't want this stuff in my tarball, so I use the `dist-hook` target to add commands that remove these unwanted files after the distribution directory has been created, but before it's "zipped" up into a tarball.

Here's a portion of the generated Makefile, showing how `dist-hook` is used by Automake:

```
...
distdir: $(DISTFILES)
        ... # copy files into distdir
        $(MAKE) $(AM_MAKEFLAGS) \
          top_distdir="$(top_distdir)" \
          distdir="$(distdir)" dist-hook
        ... # change attributes of files in distdir
...
dist dist-all: distdir
        tardir=$(distdir) && $(am__tar) | \
          GZIP=$(GZIP_ENV) gzip -c \
          >$(distdir).tar.gz
        $(am__remove_distdir)
...
.PHONY: ... dist-hook ...
...
dist-hook:
        -rm -rf `find $(distdir) -name .svn`
...
```

Don't be afraid to dig into the generated makefiles to see just exactly what Automake is doing with your code. While there's a fair amount of ugly shell code in there, most of it can be ignored. You're usually more interested in the `make` rules that Automake is generating, and these are easily separated out. Once you understand the rules, you are well on your way to becoming an Automake expert.

Designing the `ftk/src/Makefile.am` file

I've left the most difficult task for last. I now need to create appropriate `Makefile.am` files in the `src` and `utils` directories. I want to ensure that all of the original functionality is preserved from the old build system as I'm creating these files. Basically, this includes:

- properly building the ftk shared and static libraries;
- properly specifying installation locations for all installed files;
- setting the ftk library version information correctly;
- ensuring that all remaining unused files are distributed;
- ensuring that platform-specific compiler options are used.

Besides a few additions to ftk's `configure.ac` file, the following framework should cover most of the points above, so I'll be using it for all of the FLAIM library projects, with appropriate additions and subtractions, based on the needs of each individual library:

```
EXTRA_DIST = ...

lib_LTLIBRARIES = ...

include_HEADERS = ...

xxxxx_la_SOURCES = ...

xxxxx_la_LDFLAGS = -version-info x:y:z
```

The original GNU makefile told me that the library was named `libftk.so`. This is a bad name for a library on Linux, as most of the three-letter acronyms are already taken for other purposes within the file system, so I've made an executive decision here and renamed the `ftk` library to `flaimtk`. I added the libtool library name, `libflaimtk.la` to the `lib_LTLIBRARIES` list, and then changed the `xxxxx` portions of the remaining macros to `libflaimtk`.

To get the source files, I could have entered them all by hand, but I noticed while reading the original makefile that it used the GNU make function macro, `$(wildcard src/*.cpp)` in order to build the file list for the library from the contents of the `src` directory. This tells me that all of the `.cpp` files within the source directory are required by the library. To get the file list into `Makefile.am`, I used a simple shell command to concatenate the file list to the end of the `Makefile.am` file (assuming I'm in the `ftk/src` directory):

```
$ ls >> Makefile.am
```

This leaves me with a single column list of all of the files in the `ftk/src` directory appended to the bottom of the `ftk/src/Makefile.am` file. I deleted the `Makefile.am` file from this list, and then moved the list to just below the `libflaimtk_la_SOURCES =` entry. I added a BACKSLASH character after the EQUAL sign, and at the end of each of the files except the last one. This gives me a clean file list. Another formatting technique is to simply wrap the line every 70 characters or so with a BACKSLASH and a CARRIAGE RETURN. I prefer to put each file on a separate line--especially early on in the conversion process, so that I can easily extract or add files to the lists.

For the header files, I had to manually examine each one to determine its use in the project. There are only four header files in the `src` directory, and as it turns out, the only one *not* used by ftk on Unix and GNU/Linux platforms is `ftknlm.h`. This file is specific to the NetWare build. I added this file to the `EXTRA_DIST` list.

The `ftk.h` file (now renamed to `flaimtk.h`) is the only public header file, so I moved that one into the `include_HEADERS` list. The other two are used internally in the library build, so I left them in the `libflaimtk_la_SOURCES` list.

Finally, I noted in the original makefile, that the last ftk library that was released to the public in a distribution sported an interface version of 4.0.0. However, since I change the name of the library from `libftk` to `libflaimtk`, I reset this value to 0.0.0 because it's a different library now, so I replaced `x:y:z` with `0:0:0` in the `-version-info` option within the `libflaimtk_la_LDFLAGS` variable. (_NOTE: Version 0.0.0 is the default, so I could have simply removed the `-version-info` argument entirely for the same effect.)

Here's (most of) the final `ftk/src/Makefile.am` file:

```
EXTRA_DIST = ftknlm.h


lib_LTLIBRARIES = libflaimtk.la

include_HEADERS = flaimtk.h


libflaimtk_la_SOURCES = \
 ftkarg.cpp \
 ftkbtree.cpp \
 ftkcmem.cpp \
 ftkcoll.cpp \
 ...
 ftksys.h \
 ftkunix.cpp \
 ftkwin.cpp \
 ftkxml.cpp


libflaimtk_la_LDFLAGS = -version-info 0:0:0
```

That's it! You know--I don't know about you, but I'd much rather maintain this file, than a 2200 line GNU makefile! Granted, I'm not really done yet, but (trust me) it won't get much worse than this.

Moving on to the `ftk/util` directory

Properly designing a `Makefile.am` file for the `util` directory requires examining the original makefile again for more products--those built from files in the `ftk/util` directory. A quick glance at the `ftk/util` directory showed that there was only one source file, `ftktest.cpp`. This appeared to be some sort of testing program for the ftk library, so I had a design decision to make here: should I build this as a normal program, or as a "check" program.

The difference, of course, is that normal programs are always built, but "check" programs are only built when `make check` is executed. Remember also that `check` programs are never installed. Thus, if I chose to always build `ftktest`, I'd then have to decided whether or not I want it to be installed. If I want it built all the time, but not installed, I'd have to specify the program using the `noinst` prefix, rather than the usual `bin` prefix.

In either case, I probably want to add the `ftktest` binary to the list of tests run during `make check`, so the two questions here are (1) whether or not I might wish to run `ftktest` manually at times after a build, and (2) do I want to install the `ftktest` program? Given that ftk is rather mature at this point, I opted to not install `ftktest` and only build it during `make check`. Here's my final `ftk/util/Makefile.am` file:

```
FTKINC=-I$(top_srcdir)/src

FTKLIB=../src/libflaimtk.la



check_PROGRAMS = ftktest



ftktest_SOURCES = ftktest.cpp

ftktest_CPPFLAGS = $(FTKINC)

ftktest_LDADD = $(FTKLIB)



TESTS = ftktest
```

Note that I could easily have left out the `FTKINC` and `FTKLIB` variables, replacing their references with the appropriate text in the `CPPFLAGS` and `LDADD` variables, but since this will be a pattern used quite often in the new FLAIM build system, because of the external dependency between the database projects and the tool kit, I've decided to start the habit right here and now.

I hope by now that you can see the relationship between `TEST` and `check_PROGRAMS`. To be blunt, there really is *no* relationship between the files listed in `check_PROGRAMS` and those listed in `TEST`. `TEST` can refer to anything that can be executed without command line parameters, and these programs or scripts are executed during `make check` after all of the `check_PROGRAMS` are built (if any). This separation of duties makes for a very clean and flexible system.

Designing the xflaim build system

Now that I've finished with the tool kit, I'll move on to the xflaim project. I'm choosing xflaim, rather than flaim, because it supplies the most build features that can be converted to GNU Autotools, including the Java and CSharp language bindings. After xflaim, covering the remaining database projects would be redundant, as the processes are identical (but simpler). However, you can find the other build system files in the attached source archive, as usual.

I generated the `configure.ac` script using `autoscan` again. It's important to use `autoscan` in each of the individual projects, because the source code of each project is different, and will thus cause different macros to be written into each `configure.scan` file. I then used the same techniques to create xflaim's `configure.ac` script as I used with the tool kit.

The xflaim `configure.ac` script

After hand-modifying the generated `configure.scan` file and renaming it to `configure.ac`, I found this `configure.ac` script to be similar in many ways to ftk's `configure.ac` script. As it's fairly long, I'll only show you the most significant differences here:

```
...

# Checks for optional programs.

AC_PROG_TRY_CSC

AC_PROG_TRY_CSVM

AC_PROG_TRY_JAVAC

AC_PROG_TRY_JAVAH

AC_PROG_TRY_JAVADOC

AC_PROG_TRY_JAR

AC_PROG_TRY_DOXYGEN



# Configure variables: FTKLIB and FTKINC.

AC_ARG_VAR([FTKLIB], [where libflaimtk.la is at])

AC_ARG_VAR([FTKINC], [where flaimtk.h is at])



# Ensure that both or neither are specified.

if (test -n "$FTKLIB" && test -z "$FTKINC") || \

   (test -n "$FTKINC" && test -z "$FTKLIB"); then
```

```
  AC_MSG_ERROR([both or neither FTK variables])
fi

# Not specified? Check for FTK in standard places.
if test -z "$FTKLIB"; then
  # Check for flaim tool kit as a sub-project.
  if test -d "$srcdir/ftk"; then
    AC_CONFIG_SUBDIRS([ftk])
    FTKINC='$(top_srcdir)/ftk/src'
    FTKLIB='$(top_builddir)/ftk/src'
  else
    # Check for flaim tool kit as a super-project.
    if test -d "$srcdir/../ftk"; then
      FTKINC='$(top_srcdir)/../ftk/src'
      FTKLIB='$(top_builddir)/../ftk/src'
    fi
  fi
fi

# Still empty? Check for installed flaim tool kit.
if test -z "$FTKLIB"; then
  AC_CHECK_LIB([flaimtk], [ftkFastChecksum],
    [AC_CHECK_HEADERS([flaimtk.h])
     LIBS="-lflaimtk $LIBS"],
    [AC_MSG_ERROR([No FLAIM Took Kit found.])])
fi
```

```
# AC_SUBST command line variables.
if test -n "$FTKLIB"; then
  AC_SUBST([FTK_LTLIB], ["$FTKLIB/libflaimtk.la"])
  AC_SUBST([FTK_INCLUDE], ["-I$FTKINC"])
fi

# Check for Java compiler.
have_java=yes
if test -z "$JAVAC"; then have_java=no; fi
if test -z "$JAVAH"; then have_java=no; fi
if test -z "$JAR"; then have_java=no; fi
if test "x$have_java" = xno; then
  echo "-----------------------------------------"
  echo " Some Java tools not found - continuing"
  echo " without XFLAIM JNI support."
  echo "-----------------------------------------"
fi
AM_CONDITIONAL([HAVE_JAVA],
  [test "x$have_java" = xyes])

# Check for CSharp compiler.
if test -z "$CSC"; then
  echo "-----------------------------------------"
  echo " No CSharp compiler found - continuing"
  echo " without XFLAIM CSHARP support."
```

```
    echo "------------------------------------------"
  fi
  AM_CONDITIONAL([HAVE_CSHARP], [test -n "$CSC"])


  ...


  echo "
    ($PACKAGE_NAME) version $PACKAGE_VERSION
    Prefix.........: $prefix
    Debug Build....: $debug
    C++ Compiler...: $CXX $CXXFLAGS $CPPFLAGS
    Linker.........: $LD $LDFLAGS $LIBS
    FTK Library....: ${FTKLIB:-INSTALLED}
    FTK Include....: ${FTKINC:-INSTALLED}
    CSharp Compiler: ${CSC:-NONE} $CSCFLAGS
    CSharp VM......: ${CSVM:-NONE}
    Java Compiler..: ${JAVAC:-NONE} $JAVACFLAGS
    JavaH Utility..: ${JAVAH:-NONE} $JAVAHFLAGS
    Jar Utility....: ${JAR:-NONE} $JARFLAGS
    Javadoc Utility: ${JAVADOC:-NONE}
    Doxygen........: ${DOXYGEN:-NONE}
  "
```

First, you'll notice that I've invented a few more of my `AC_PROG_TRY` macros. In the first portion, I'm checking for the optional existence of the following programs: a CSharp compiler, a CSharp virtual machine, a Java compiler, a JNI header and stub generator, a javadoc generation tool, a Java archive tool, and of course, doxygen. As before, I've written separate macro files for each of these checks, and added them to my `xflaim/m4` directory.

As with the `AC_PROG_TRY_DOXYGEN` macro, each of these macros attempts to locate the associated program, but doesn't go apoplectic if it's not found, because I want to be able to use the program if it's there, but not require my users to have them in order to build some of the most useful functionality of the FLAIM projects.

Next, you'll find a new macro, `AC_ARG_VAR`. Like the `AC_ARG_ENABLE` and `AC_ARG_WITH` macros, `AC_ARG_VAR` allows the project maintainer to extend the interface to the `configure` script. This variable is different, however, in that it adds a public variable to the list of variables that the configure script cares about. In this case, I'm adding two public variables, `FTKINC` and `FTKLIB`. These variables will show up in the `configure` script's help text under the section entitled "Some influential environment variables:".

These variables are also automatically substituted into the `Makefile.in` templates generated by Automake. However, I don't really need this substitution functionality, as I'm going to build other variables out of these variables, and I'll want these derived variables to be substituted, as you'll soon see.

These variables may be set by the user in the environment, or specified on the `configure` script's command line in this manner:

```
$ ./configure FTKINC='$HOME/dev/ftk/include' ...
```

The large chunk of code that follows the `AC_ARG_VAR` macros actually uses these variables to set other variables used in the build system:

```
...

# Ensure that both or neither are specified.

if (test -n "$FTKLIB" && test -z "$FTKINC") || \

   (test -n "$FTKINC" && test -z "$FTKLIB"); then

  AC_MSG_ERROR([both or neither FTK variables])

fi



# Not specified? Check for FTK in standard places.

if test -z "$FTKLIB"; then

  # Check for flaim tool kit as a sub-project.

  if test -d "$srcdir/ftk"; then

    AC_CONFIG_SUBDIRS([ftk])

    FTKINC='$(top_srcdir)/ftk/src'

    FTKLIB='$(top_builddir)/ftk/src'

  else
```

```
    # Check for flaim tool kit as a super-project.
    if test -d "$srcdir/../ftk"; then
      FTKINC='$(top_srcdir)/../ftk/src'
      FTKLIB='$(top_builddir)/../ftk/src'
    fi
  fi
fi

# Still empty? Check for installed flaim tool kit.
if test -z "$FTKLIB"; then
  AC_CHECK_LIB([flaimtk], [ftkFastChecksum],
    [AC_CHECK_HEADERS([flaimtk.h])
     LIBS="-lflaimtk $LIBS"],
    [AC_MSG_ERROR([No FLAIM Took Kit found.])])
fi

# AC_SUBST command line variables.
if test -n "$FTKLIB"; then
  AC_SUBST([FTK_LTLIB], ["$FTKLIB/libflaimtk.la"])
  AC_SUBST([FTK_INCLUDE], ["-I$FTKINC"])
fi
...
```

First, I check to see that either both variables are specified, or neither. If only one of them is given, then I have to fail with an error. The user isn't allowed to tell me where to find half the tool kit. I need both the include file and the library.

If neither is specified, then I go searching for them. First I look for a sub-directory called `ftk`. If I find one, then I configure that directory as a sub-project to be processed by Autoconf, by using the `AC_CONFIG_SUBDIRS` macro. Note that you can use this macro

conditionally, and multiple times within the same `configure.ac` file. I also set the variables to point to the appropriate relative locations within the ftk project.
If I don't find it as a sub-directory, then I look for it in the parent directory. If I find it there, I set the `FTK` variables appropriately. This time I don't need to configure the located `ftk` directory as a sub-project, because I'm assuming that the *current* project (xflaim) is already a sub-project of the umbrella project.
If I don't find it in either place, I use the standard `AC_CHECK_LIB` and `AC_CHECK_HEADERS` macros to see if it's installed on the user's host machine. If so, I need only add `-lflaimtk` to the `$LIBS` variable. The header file will be found in the standard location--usually `/usr(/local)/include`. Note that normally, `AC_CHECK_LIB` would automatically add the library reference to the `$LIBS` variable, but since I've overridden the default functionality in the third parameter, I have to add it myself.
If I don't find it installed, then I give up with an error message, indicating that xflaim can't be built without the FLAIM tool kit.

However, after making it through the checks, if the `FTKLIB` variable is no longer empty, then I use `AC_SUBST` to "publish" `FTK_INCLUDE` and `FTK_LTLIB` variables, containing derivations of the `FTK` variables appropriate for the C++ preprocessor and the linker.
The remaining code (excluding the trailing `echo` statement) calls `AM_CONDITIONAL` for Java and CSharp tools in a manner similar to the way I handled doxygen. Again, I generate bold messages to the user that the Java or CSharp portions of the xflaim project will not be built if those tools can't be found, but I allow the build to continue.

Creating the `xflaim/src/Makefile.am` file

I wrote the `xflaim/src/Makefile.am` file by following the same design principles used in the `ftk/src` version of that file. It looks very similar to its ftk counterpart, with one exception: According to the original build system makefile, the Java native interface (JNI) and CSharp native language binding sources are compiled and linked right into the xflaim shared library.
This is not an uncommon practice, because it alleviates the need for extra library objects specifically for these languages. Essentially, the xflaim shared library exports native interfaces for these languages, that are then consumed by their corresponding language binding wrappers.

I'm going to ignore these language binding interfaces for now. However, keep them in the back of your mind, because later when I've finished with the entire xflaim project, I'll turn my attention back to properly hooking these bindings into the library. Except for the language bindings then, the `Makefile.am` file looks almost identical to its ftk counterpart:

```
SUBDIRS =


if HAVE_JAVA
  SUBDIRS += java
  JNI_LIBADD=java/libxfjni.la
endif
```

```
if HAVE_CSHARP

  SUBDIRS += cs

  CSI_LIBADD=cs/libxfcsi.la

endif


SUBDIRS += .


lib_LTLIBRARIES = libxflaim.la

include_HEADERS = xflaim.h


libxflaim_la_SOURCES = \

 btreeinfo.cpp \

 f_btpool.cpp \

 f_btpool.h \

 ...

 rfl.h \

 scache.cpp \

 translog.cpp


libxflaim_la_CPPFLAGS = $(FTK_INCLUDE)

libxflaim_la_LIBADD = $(JNI_LIBADD)\

 $(CSI_LIBADD) $(FTK_LTLIB)

libxflaim_la_LDFLAGS = -version-info 3:2:0
```

As I did with the `docs` directory in the top-level `Makefile.am` file, I've conditionally defined the `SUBDIRS` variable here, based on the Automake conditional specified in `configure.ac`. What's different here is that I've pre-defined `SUBDIRS` to be empty before checking the condition, and then added the current directory (`.`) at the end. These directories must be processed (if they can be) before the current directory, as they generate libraries that must be linked into the library built by this makefile. I had to

initialize `SUBDIRS` to empty because the PLUS-EQUAL (`+=`) Automake extension operator will only work properly if the variable is already defined--even if it must be defined as empty.
Since I initialized it to empty, I removed the implicit current directory, so I added it back in after the conditional checks. It's a bit clumsy, I know, but it works.

The library interface version information was once again extracted from the original xflaim project makefile.

Turning to the `xflaim/util` directory

The `util` directory for xflaim is a bit more complex. According to the original makefile, it generates several utility programs, as well as a convenience library that is consumed by each of these utilities.
In addition, the task of finding out which source files belong to which utilities, and which were not used at all was more difficult. It turns out that there are several files in the `xflaim/util` directory that are not used by any of the utilities. I suppose the project developers thought there might be some future value in these source files, so they kept them around. Well, that leaves us with another decision: Do we distribute these "extra" source files? I chose to do so, as they were already being distributed by the original build system, and adding them to the `EXTRA_DIST` list makes it obvious to later observers that they aren't used in the build.
Here's my version of the `xflaim/util/Makefile.am` file:

```
EXTRA_DIST = dbdiff.cpp dbdiff.h domedit.cpp\
 diffbackups.cpp xmlfiles

XFLAIM_INCLUDE=-I$(top_srcdir)/src
XFLAIM_LDADD=../src/libxflaim.la

## Utility Programs

bin_PROGRAMS = xflmcheckdb xflmrebuild\
 xflmview xflmdbshell

xflmcheckdb_SOURCES = checkdb.cpp
xflmcheckdb_CPPFLAGS =\
 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)
xflmcheckdb_LDADD = libutil.la $(XFLAIM_LDADD)
```

```
xflmrebuild_SOURCES = rebuild.cpp
xflmrebuild_CPPFLAGS =\
 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)
xflmrebuild_LDADD = libutil.la $(XFLAIM_LDADD)

xflmview_SOURCES = \
 viewblk.cpp \
 view.cpp \
 ...
 viewmenu.cpp \
 viewsrch.cpp

xflmview_CPPFLAGS =\
 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)
xflmview_LDADD = libutil.la $(XFLAIM_LDADD)

xflmdbshell_SOURCES = \
 domedit.h \
 fdomedt.cpp \
 fshell.cpp \
 fshell.h \
 xshell.cpp

xflmdbshell_CPPFLAGS =\
 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)
```

```
xflmdbshell_LDADD = libutil.la $(XFLAIM_LDADD)

## Utility Convenience Library

noinst_LTLIBRARIES = libutil.la

libutil_la_SOURCES = \
 flm_dlst.cpp \
 flm_dlst.h \
 flm_lutl.cpp \
 flm_lutl.h \
 sharutil.cpp \
 sharutil.h

libutil_la_CPPFLAGS =\
 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)

## Check Programs

check_PROGRAMS = \
 ut_basictest \
 ut_binarytest \
 ...
 ut_xpathtest \
 ut_xpathtest2
```

```
check_DATA = copy-xml-files.stamp

check_HEADERS = flmunittest.h


ut_basictest_SOURCES =\

 flmunittest.cpp basictestsrv.cpp

ut_basictest_CPPFLAGS =\

 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)

ut_basictest_LDADD = libutil.la $(XFLAIM_LDADD)


...


ut_xpathtest2_SOURCES =\

 flmunittest.cpp xpathtest2srv.cpp

ut_xpathtest2_CPPFLAGS =\

 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)

ut_xpathtest2_LDADD = libutil.la $(XFLAIM_LDADD)


## Unit Tests


TESTS = \

 ut_basictest \

 ...

 ut_xpathtest2


## Miscellaneous rules required by Check Programs
```

```
copy-xml-files.stamp:

        cp $(srcdir)/xmlfiles/*.xml .

        echo Timestamp > $@



clean-local:

        -rm -rf ix2.*

        -rm -rf bld.*

        -rm -rf tst.bak

        -rm -f *.xml

        -rm -f copy-xml-files.stamp
```

In this example, you can see by the ellipses that I left out several long lists of files and products. There are, for instance, 22 unit tests built by this makefile. I only left the descriptions for two of them, because they're all identical, except for naming differences and the source files from which they're built.

But here's something curious. Take a look at the definition for the `xflmcheckdb` program:

```
...

xflmcheckdb_SOURCES = checkdb.cpp

xflmcheckdb_CPPFLAGS =\

 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)

xflmcheckdb_LDADD = libutil.la $(XFLAIM_LDADD)

...
```

Notice that the `xflmcheckdb_CPPFLAGS` variable uses both the `XFLAIM_INCLUDE` and `FTK_INCLUDE` variables. The utility clearly requires information from both sets of header files. But the `xflmcheckdb_LDADD` variable only uses the `XFLAIM_LDADD` variable. Why? Because Libtool manages inter-library dependencies for you. Since I reference `libxflaim.la` (through `XFLAIM_LDADD`) when building the utilities and unit tests, and since `libxflaim.la` lists `libflaimtk.la` as a dependency, I don't need to explicitly reference that library here.
You can get a clearer picture of this if you take a look at the contents of `libxflaim.la` (in your build directory under `xflaim/src`). You'll find a few lines like this somewhere in the middle of the file:

```
...

# Libraries that this one depends upon.

dependency_libs=

 ' .../flaim/build/ftk/src/libflaimtk.la

 -lrt -lpthread -lncurses'

...
```

Notice that the path information for `libflaimtk.la` is listed here, thus we don't have to specify it in the `LDADD` variables for the xflaim utilities. The linker still requires this information, but the `libtool` script effectively hides this requirement by extracting the information from the `.la` file and appending it to the linker command line when building the utility files.

As an aside, when `libxflaim.la` is installed, Libtool modifies the installed version of this file such that it references the installed versions of the libraries, rather than those in the build directory structure.

Stamp targets

In creating this makefile, I ran across another minor problem that I hadn't anticipated. At least one of the unit tests (probably several) seemed to require that some XML data files be present in the directory from which the test is run. What brought this to my attention was the fact that that particular unit test failed. When I dug into it, I noticed that it was trying to open some specifically named XML data files. Searching around a bit lead me to the `xmldata` directory, beneath the `xflaim/util` directory. This directory contained several dozen XML data files.

Somehow I needed to copy those files into the build hierarchy's `xflaim/util` directory before I could run the unit tests. Well, I know that `check` programs are built before `TESTS` are executed. As it turns out other primaries prefixed with `check` are also processed before `TESTS` are executed. Notice the `check_DATA` variable:

```
...

check_DATA = copy-xml-files.stamp

...

copy-xml-files.stamp:

        cp $(srcdir)/xmlfiles/*.xml .

        echo Timestamp > $@

...
```

It refers to a file called `copy-xml-files.stamp`. This is a special type of file target called a "stamp" target. It's purpose is to replace a bunch of unspecified files, or a non-file-based operation, with one single representative file. This stamp file is used to indicate to the make system that the operation of copying all of the XML data files into the test

directory has been done. Automake uses stamp files quite often in its own generated rules.
The rule for generating the stamp file (near the bottom of the example above), also copies the XML data files into the test execution directory. The `echo` statement simply creates a file named `copy-xml-files.stamp`, and containing the single word, "Timestamp". The file may contain anything, really. The important point here is that the file exists, and has a time and date associated with it. The `make` utility uses this information to determine whether or not the copy operation needs to be executed. In this case, since `copy-xml-files.stamp` has no dependencies, its mere existence indicates to `make` that the operation has already been done, and need not be done again. To get `make` to perform the copy operation on the next build, simply delete the stamp file.
This is a sort of hybrid between a true file-based rule, and a phony target. Phony targets are always executed, because they aren't real files, so `make` has no way of knowing whether or not the associated operation *should* be performed. The time stamps of file-based rules can be checked against their dependency lists to determine if they should be re-executed, or not. Stamp rules like this one are executed only if the stamp file is missing.
All files placed in the build directory should be cleaned up when the user enters `make clean` at the command prompt. Since I placed these XML data files into this directory, I need to clean them up also. Files listed in `DATA` variables are not cleaned up automatically, because `DATA` files are usually not generated files. Most often, the `DATA` primary is used to list existing project files that need to be installed. In this case, I actually created a bunch of XML files and a stamp file, so I need to clean these up when `make clean` is executed.
NOTE: Be careful when using this technique on files that need to be copied from the source directory into the same corresponding location in the build directory. Special care needs to be taken to ensure you don't inadvertently delete source files from the source tree when building in the source tree.

Cleaning your room

There is another way to ensure that files created using your own `make` rules get cleaned up during execution of the `clean` target. You may also define the `CLEANFILES` variable to contain a white space separated list of files or wild card specifications to be removed. The `CLEANFILES` variable is the more "approved" method of removing extra files during `make clean`.
If that's so, then why did I use `clean-local` in this case? Because the `CLEANFILES` variable has one caveat: it won't remove directories, only files. Each of the `rm` commands above removes a wild card file specification that contains at least one directory, so I had to use `clean-local` in this case. I'll show you a proper use of `CLEANFILES` shortly.

```
...

clean-local:

        -rm -rf ix2.*

        -rm -rf bld.*

        -rm -rf tst.bak

        -rm -f *.xml

        -rm -f copy-xml-files.stamp
```

```
...
```

Here, I needed to remove all files ending in `.xml`, plus the stamp file. In addition, the unit tests themselves are not well written, in that they leave "droppings" behind. Let this be a lesson: when you write unit tests that generate files and directories, remove all such droppings before terminating your test. That way, you won't have to write such `clean` rules in your makefiles.

Another way of managing this is would be to write a script that calls the tests, and then cleans up left-over files and directories. This script then becomes the entry in the `TESTS` variable.

I use the Automake supported `clean-local` target here as a way to extend the `clean` target. The `clean-local` target is executed as a dependency of (and thus before) the `clean` target, if it exists. Here's the corresponding code from the Automake-generated `Makefile`:

```
...

clean: clean-am



clean-am: clean-binPROGRAMS clean-checkPROGRAMS \

        clean-generic clean-libtool clean-local \

        clean-noinstLTLIBRARIES mostlyclean-am

...

.PHONY: ... clean-local ...

...

clean-local:

        -rm -rf ix2.*

        -rm -rf bld.*

        -rm -rf tst.bak

        -rm -f *.xml

        -rm -f copy-xml-files.stamp

...
```

Automake noted that I had a target named `clean-local` in my `Makefile.am` file, so it added `clean-local` to the dependency list for `clean-am`, and then added it to the `.PHONY` list. Had I not written a `clean-local` target, these references would have been missing from the generated `Makefile`.

When cleaning up files in a build directory using wild cards in this manner, you need to remember that the user may be building in the source directory. Try to make your wild cards as specific as possible so you don't inadvertently remove source files.

Building Java sources using Autotools

The most significant barrier to building Java sources using the GNU Autotools is the (apparently nearly intentional) misdirection in the existing documentation. Now, I know better than to think it was done on purpose, but time and time again, what you find in internet searches, or in the GNU Automake documentation is just enough information, presented in just such a way as to allow you to really hang yourself well when you try to use it. There's nothing quite as frustrating as finding dozens of implications that something can be done, but finding no information telling you exactly how to do it.

There are two sections in the GNU Automake manual that refer to building Java sources using the GNU Autotools. The first is section 8.15, entitled, "Java Support". The second is section 10.4, entitled simply, "Java". (The major section 10 is entitled, "Other GNU Tools".)

In the first place, the contents of these two sections should probably be swapped. Section 8.15 actually discusses using the GCJ front end to the GNU compiler suite to compile and link Java source code into *native executables*. This is nothing that the average Java purist would understand without a little hand-holding, because Sun Java doesn't do anything of the sort. The information in this section would be better placed under a section entitled, "Other GNU Tools" (like section 10, for instance).
On the other hand, section 10.4 talks about building Java sources using whatever `javac` compiler happens to be found in the system path. This is much more likely to be something a Java developer might actually wish to do in a `Makefile.am` file, so I'm going to ignore section 8.15 (native compilation, using GCJ), and talk strictly about section 10.4.

Autotools Java support

Autoconf has *no* built-in support for java. For example, it provides no macros that locate Java tools in the end user's environment. Automake's support for building Java classes is minimal, but getting it to work is not that difficult if you know what you're doing. Automake provides a built-in primary (`JAVA`) for building Java sources. Automake does not provide any preconfigured installation location prefixes for installing Java classes. However, the usual place to install Java classes and `.jar` files is in the `$(datadir)/java` directory. So, creating a proper prefix is as simple as using the Automake prefix extension mechanism of defining a variable suffixed with `dir`:

```
...

javadir = $(datadir)/java

java_JAVA = file_a.java file_b.java ...

...
```

Note that you don't often want to install Java sources, which is what you will accomplish when you define your `JAVA` primary with this sort of prefix. Rather, you want the class files to be installed, or more likely a `.jar` file containing all of your `.class` files. So I find it more useful to define the JAVA primary with the `noinst` prefix. Additionally, files in the `JAVA` primary list are not distributed by default, so you may even want to use the `dist` super-prefix, in this manner:

```
...

dist_noinst_JAVA = file_a.java file_b.java ...

...
```

When you define a list of Java source files in a variable containing the `JAVA` primary, Automake generates a make rule that builds that list of files all in one command, using the following command line syntax:

```
...

JAVAROOT = $(top_builddir)

JAVAC = javac

CLASSPATH_ENV = CLASSPATH=$(JAVAROOT):\

  $(srcdir)/$(JAVAROOT):$$CLASSPATH

...

classdist_noinst.stamp: $(dist_noinst_JAVA)

        @list1='$?'; list2=; \

        if test -n "$$list1"; then \

          for p in $$list1; do \

            if test -f $$p;

              then d=; \

              else d="$(srcdir)/"; \

            fi; \

            list2="$$list2 $$d$$p"; \

          done; \

          echo '$(CLASSPATH_ENV) $(JAVAC) \

            -d $(JAVAROOT) $(AM_JAVACFLAGS) \
```

```
            $(JAVACFLAGS) '"$$list2"; \
          $(CLASSPATH_ENV) $(JAVAC) \
            -d $(JAVAROOT) $(AM_JAVACFLAGS) \
            $(JAVACFLAGS) $$list2; \
        else :; fi
        echo timestamp > classdist_noinst.stamp
...
```

Most of the "stuff" you see in the command above is for prepending the `$(srcdir)` prefix onto each file in the user-specified list, in order to properly support `VPATH` builds. This code uses a shell `for` statement to split the list into individual files, prepend `$(srcdir)`, and then reassemble the list.

NOTE: It's interesting to note that this file list munging process could have been done in a half-line of GNU `make`-specific code, but Automake is designed to generate makefiles that can be executed by many older make programs.

The part that actually does the work is found in one line, near the bottom. To make it simpler to read, I'll reformat this example, removing the cruft:

```
...
JAVAROOT = $(top_builddir)
JAVAC = javac
CLASSPATH_ENV = CLASSPATH=$(JAVAROOT):\
  $(srcdir)/$(JAVAROOT):$$CLASSPATH
...
classdist_noinst.stamp: $(dist_noinst_JAVA)
        ...
        $(CLASSPATH_ENV) $(JAVAC) -d $(JAVAROOT) \
          $(AM_JAVACFLAGS) $(JAVACFLAGS) $$list
...
```

You may have noticed Automake's use of a stamp file here. This is done because the single `$(JAVAC)` command generates several `.class` files from several `.java` files. Rather than just pick one of these at random to use in the rule, Automake generates and uses a stamp file. This is important to know, because using a stamp file in the rule causes `make` to ignore the associations between individual `.class` files and their corresponding `.java` files. That is, if you delete a `.class` file, the rules in the `Makefile` will not cause it to be rebuilt. The only way to cause the re-execution of the `$(JAVA)` command is to either modify one or more of the `.java` files, thereby causing their timestamps to become newer than that of the the stamp file, or to delete the stamp file entirely.

The variables used in the build environment, and on the command line include `JAVAROOT`, `JAVAC`, `JAVACFLAGS`, `AM_JAVACFLAGS` and `CLASSPATH_ENV`. Each of these may be specified by the developer in the `Makefile.am` file. If they're not specified, then the defaults you see in this example are used instead. Where you don't see a default value set, you may assume the default value is empty.

One important point about this code is that all of the files specified in the `JAVA` primary list are compiled using a single command line, which could pose a problem on systems with limited command line lengths. If you find you have such a problem, you may have to develop your own make rules for building Java classes. Given the limited support that Automake currently provides, this isn't really a very daunting task.

The `CLASSPATH_ENV` variables sets the Java `classpath` environment variable for the `javac` command such that it contains the contents of `JAVAROOT` (`$(top_builddir)`, by default), the same value prefixed with `$(srcdir)`, and then any class path that might be specified in the environment by the user.

The `JAVAC` variable contains `javac` by default. The hope here is that `javac` can be found in the system path. The `AM_JAVACFLAGS` variable may be set in the `Makefile.am` file by the developer. As usual, the non-Automake version of this variable (`JAVACFLAGS`) is considered a "user" variable, and shouldn't be set in makefiles.

The `JAVAROOT` variable is used to specify the location of the java root directory, which is where the Java compiler will expect to find the start of packages directory hierarchies belonging to your project.

This is fine as far as it goes, but it doesn't go nearly far enough. In this (relatively simple) project, I also need to generate JNI header files using the `javah` utility, and I need to generate a `.jar` file from the `.class` files built from my Java sources. Automake-provided Java support doesn't even begin to handle these tasks. So I'll have to do the rest with hand-coded `make` rules. I'll start with Autoconf macros to ensure that I have a good Java build environment.

Using ac-archive macros

I did a little hunting around on the internet, and found that the ac-archive project on sourceforge.net does in fact supply Autoconf macros that come close to what I need in order to ensure that I have a good Java development environment. I downloaded the latest ac-archive source package, and just hand-installed the `.m4` files that I needed into my `xflaim/m4` directory.

Then I modified them (and their names) such that they work the way my `AC_PROG_TRY_DOXYGEN` macro works. I wanted to locate Java tools if they exist, but be able to continue without them if they're missing. Given the current politics surrounding the existence of Java tools in GNU/Linux distributions at this time, this is probably a wise approach.

NOTE: The other way to use the ac-archive package is to actually install it on your system, which will place the ac-archive `.m4` files into the `/usr/(local/)share/ac-archive` directory. The documentation for ac-archive provides instructions on how you might pass flags to the `aclocal` utility from within your project's top-level `Makefile.am` file that tell it how to access the installed ac-archive macros during an execution of `autoreconf`, or `aclocal`.

I created the following macros and files from those found in the ac-archive:

- `AC_PROG_TRY_JAVAC` is defined in `ac_prog_try_javac.m4` and `ac_prog_javac_works.m4`

- `AC_PROG_TRY_JAVAH` is defined in `ac_prog_try_javah.m4`
- `AC_PROG_TRY_JAVADOC` is defined in `ac_prog_try_javadoc.m4`
- `AC_PROG_TRY_JAR` is defined in `ac_prog_try_jar.m4`
- `AC_PROG_TRY_CSC` is defined in `ac_prog_try_csc.m4` and `ac_prog_csc_works.m4`
- `AC_PROG_TRY_CSVM` is defined in `ac_prog_try_csvm.m4` and `ac_prog_csvm_works.m4`

With only a little more effort, I was also able to create the CSharp macros I needed to accomplish the same tasks for the CSharp language bindings. I'll discuss CSharp in the next section. Here's a portion of the xflaim `configure.ac` file, repeated here for your information:

```
...

# Checks for optional programs.

AC_PROG_TRY_CSC



AC_PROG_TRY_CSVM

AC_PROG_TRY_JAVAC

AC_PROG_TRY_JAVAH

AC_PROG_TRY_JAVADOC

AC_PROG_TRY_JAR

...

# Check for Java compiler.

have_java=yes

if test -z "$JAVAC"; then have_java=no; fi

if test -z "$JAVAH"; then have_java=no; fi

if test -z "$JAR"; then have_java=no; fi

if test "x$have_java" = xno; then

  echo "-----------------------------------------"

  echo " Some Java tools not found - continuing"

  echo " without XFLAIM JNI support."

  echo "-----------------------------------------"
```

```
fi

AM_CONDITIONAL([HAVE_JAVA],

  [test "x$have_java" = xyes])



# Check for CSharp compiler.

if test -z "$CSC"; then

  echo "-----------------------------------------"

  echo " No CSharp compiler found - continuing"

  echo " without XFLAIM CSHARP support."

  echo "-----------------------------------------"

fi

AM_CONDITIONAL([HAVE_CSHARP], [test -n "$CSC"])

...
```

These macros set the `CSC`, `CSVM`, `JAVAC`, `JAVAH`, `JAVADOC` and `JAR` variables to the location of their respective CSharp and Java tools, and then substitute them into the xflaim project's `Makefile.in` templates using `AC_SUBST`. If any of these variables are already set in the user's environment when the `configure` script is executed, their values are left untouched, allowing the user to override the values that would have been set by the macros.

I also added some shell code to set a variable, `have_java` to either `yes` or `no`, depending on whether or not all three tools could be found. If they are found, `have_java` becomes `yes`, which fact is later used in the call to `AM_CONDITIONAL`. Recall that this Automake macro conditionally sets the `HAVE_JAVA` variable, which is later used in `xflaim/src/Makefile.am` file to conditionally build the `java` sub-directory hierarchy.

Canonical system information

The only non-obvious bit of information you need to know about using these ac-archive extensions is that they rely on the built-in Autoconf macro, `AC_CANONICAL_TARGET`. Autoconf provides a way to automatically expand any existing macros inside the definition of a macro, so that macros required by the one being defined can be made available immediately. However, if `AC_CANONICAL_TARGET` is not used before certain other macros (including, unfortunately, `LT_INIT`), then `autoreconf` will generate about a dozen warning messages.

To alleviate these warnings, I added `AC_CANONICAL_SYSTEM` to my top-level and xflaim-level `configure.ac` files, immediately after the call to `AC_INIT`. As I mentioned earlier in this chapter, this macro and those that it calls, `AC_CANONICAL_BUILD`, `AC_CANONICAL_HOST` and `AC_CANONICAL_TARGET`, are designed to ensure that the `$host`, `$build` and `$target` environment variables are defined by the `configure` script, such that they contain appropriate values describing the user's host, build and target systems.

These variables contain canonical values for the host, build and target CPU, vendor and operating system. Values like these are very useful to extension macros. If a macro can assume these variables are set properly, then it saves quite a bit of code duplication in the macro definition.

The values of these variables are calculated using two helper scripts, `config.guess` and `config.sub`, which are distributed with Autoconf. The `config.guess` script uses a combination of `uname` commands to ferret out information about the host system, and munge it into a canonical value. The `config.sub` script is used to reformat host, build and target information specified by the user on the `configure` command line into a canonical value.

The key point here, however, is that I had to use the `AC_CANONICAL_SYSTEM` macro well before I called the ac-archive extension macros in my configure.ac script.

The `xflaim/java` directory structure

The original source layout had the Java JNI and CSharp native sources located in entirely different directory structures than `xflaim/src`. The JNI sources were located in `xflaim/java/jni`, and the CSharp native sources were located in `xflaim/csharp/xflaim`. While Automake has no problem generating rules for accessing files well outside the current directory hierarchy, I find it a bit silly to put these files so far away from the only library they can really belong to. Thus, I broke my own rule of thumb about not rearranging files in this case. I moved the contents of these two directories to directories under `xflaim/src`. I named the JNI directory `xflaim/src/java` and the CSharp native sources directory `xflaim/src/cs`.

```
flaim
  xflaim
    src
      cs
      java
        wrapper
          xflaim
```

As you can see, I also added a `wrapper` directory beneath the `java` directory, in which I rooted the xflaim wrapper package hierarchy. Since the Java xflaim wrapper classes are part of the Java xflaim package, they have to be located in a directory called `xflaim`. Nevertheless, the build happens in the `wrapper` directory. There are no build files found in the `wrapper/xflaim` directory, or any directories below that point.

Note that it doesn't matter how deep your package hierarchy is. You will still build the java classes in the wrapper directory--this is the `JAVAROOT` directory for this project.

The `xflaim/src/Makefile.am` file

At this point the `configure.ac` script is doing about all it can for me to ensure that I have a good Java build environment. If I have a good Java build environment, my build system will be able to generate my JNI wrapper classes and header files, and build my C++ JNI sources. If my end user's system doesn't provide these tools, then she simply can't build or link in the JNI language bindings on that host.

Have a look at the `xflaim/src/Makefile.am` file, and examine the portions that are relevant to building the Java and CSharp language bindings:

```
SUBDIRS =


if HAVE_JAVA

  SUBDIRS += java

  JNI_LIBADD=java/libxfjni.la

endif


if HAVE_CSHARP

  SUBDIRS += cs

  CSI_LIBADD=cs/libxfcsi.la

endif


SUBDIRS += .

...

libxflaim_la_LIBADD =\

 $(JNI_LIBADD) $(CSI_LIBADD) $(FTK_LTLIB)

...
```

I've already explained the use of the conditionals to ensure that the `java` and `cs` directories only get built if the proper conditions are met. You can now see how this fits into the build system I've created so far.

Notice that I'm also conditionally defining two new library variables. If I can build the Java language bindings, then the `JNI_LIBADD` variable will refer to the library that is built in the `java` directory. If I can build the CSharp language bindings, then the `CSI_LIBADD` variable will refer to the library that is built in the `cs` directory. In either case, if the required tools are not found by the configure script, then the associated variable will remain undefined. When an undefined variable is referenced, it expands to nothing, so there's no harm in using it in the `libxflaim_la_LIBADD` variable.

Building the JNI C++ sources

Now, allow me to turn your attention to the `xflaim/src/java/Makefile.am` file:

```
SUBDIRS = wrapper

XFLAIM_INCLUDE=-I$(srcdir)/..

noinst_LTLIBRARIES = libxfjni.la

libxfjni_la_SOURCES = \
 jbackup.cpp \
 jdatavector.cpp \
 jdb.cpp \
 jdbsystem.cpp \
 jdomnode.cpp \
 jistream.cpp \
 jniftk.cpp \
 jniftk.h \
 jnirestore.cpp \
 jnirestore.h \
 jnistatus.cpp \
 jnistatus.h \
 jostream.cpp \
 jquery.cpp

libxfjni_la_CPPFLAGS =\
 $(XFLAIM_INCLUDE) $(FTK_INCLUDE)
```

Again, I want the `wrapper` directory to be built first, because it will build the class files and JNI header files required by the JNI convenience library sources. This time, it's not conditional. If I've made it this far into the build hierarchy, then I know I have all the Java

tools I need. This `Makefile.am` file simply builds a convenience library containing my JNI C++ interface functions.
Because of the way Libtool builds both shared and static libraries from the same sources, this convenience library will become part of both the xflaim shared and static libraries. The original build system makefile accounted for this by linking the JNI and CSharp native interface objects into only the shared library.

The fact that these libraries are added to both the shared and static xflaim libraries is not really a problem. Objects in a static library remain unused in applications or libraries linking to the static library, as long as code in those objects remain unreferenced. However, I'll admit that it's a bit of a "wart" on the side of my new build system.

The Java wrapper classes and JNI headers

Finally, the `xflaim/src/java/wrapper/Makefile.am` file takes us to the heart of the matter. I've tried many different configurations for building Java JNI wrappers, and this one always comes out on top. Here's the `wrapper` directory's Automake intput file:

```
JAVAROOT = .


jarfile = $(PACKAGE)jni-$(VERSION).jar

jardir = $(datadir)/java

pkgpath = xflaim

jhdrout = ..


$(jarfile): $(dist_noinst_JAVA)

        $(JAR) cf $(JARFLAGS) $@ $(pkgpath)/*.class


jar_DATA = $(jarfile)


java-headers.stamp: $(dist_noinst_JAVA)

        @list="`echo $(dist_noinst_JAVA) |\

         sed -e 's|\.java||g' -e 's|/|.|g'`"; \

        for class in $$list; do \

          echo "$(JAVAH) -jni -d $(jhdrout)\
```

```
            $(JAVAHFLAGS) $$class"; \
           $(JAVAH) -jni -d $(jhdrout)\
            $(JAVAHFLAGS) $$class; \
         done
         @echo "JNI headers generated"\
          > java-headers.stamp


all-local: java-headers.stamp


CLEANFILES = $(jarfile) $(pkgpath)/*.class\
 java-headers.stamp $(jhdrout)/xflaim_*.h


dist_noinst_JAVA = \
 $(pkgpath)/BackupClient.java \
 $(pkgpath)/Backup.java \
 ...
 $(pkgpath)/XFlaimException.java \
 $(pkgpath)/XPathAxis.java
```

I've set the `JAVAROOT` variable to DOT (`.`), mainly because I want Automake to be able to tell the Java compiler that this is where the package hierarchy begins. The xflaim Java wrapper classes are found in the `xflaim` package. The default value for `JAVAROOT` is `$(top_builddir)`, which would have the wrapper class belong to the `xflaim.src.java.wrapper.xflaim` package. That's not right.

I then created a variable called `jarfile`, deriving its value from `$(PACKAGE)` and `$(VERSION)`. This is how the `destdir` variable is derived also, from which the name of the tarball comes. A `make` rule indicates how the `.jar` file should be built. Here, I'm using the `JAR` variable, whose value was calculated for me by the results of the `AC_PROG_TRY_JAR` macro in the `configure` script. This rule is fairly straight forward.

I've defined a new installation variable called `jardir`--the place where `.jar` files are to be installed, presumably. And I've used it as the prefix for a `DATA` primary. Any files that Automake doesn't understand--basically, any files that you build using your own rules--are just considered by Automake to be data files, and are installed as such.

I'm using another stamp file in the rule that builds the JNI header files from the `.class` files. I'm doing this for the same reason that Automake used a stamp file in the rule that it uses to build `.class` files from `.java` source files.

This is the most complex part of this makefile, so I'll try to break it into simple pieces. The rule states that the stamp file depends on the files listed in the `dist_noinst_JAVA` variable. The command is a bit of complex shell script that strips the `.java` extensions from the file list, and converts all the SLASH characters in to DOT characters. The reason for this is that the `javah` utility wants a list of class names, not a list of file names. The last line, of course, generates the stamp file.

Finally, I hooked my `java-headers.stamp` target into the `all` target by adding it as a dependency to the `all-local` target. When the `all` target (the default for all Automake-generated makefiles) is executed in this makefile, `java-headers.stamp` will be built, along with the JNI headers.

Here, I've also added the `.jar` file, all of the `.class` files, the `java-headers.stamp` file and all of the generated JNI header files to the `CLEANFILES` variable, so that Automake will clean them up for me when `make clean` is executed on this makefile. Again, I can use the `CLEANFILES` variable here because I'm not trying to delete any directories.

A caveat about using the JAVA primary

There's one important caveat to using the `JAVA` primary. You may only define one `JAVA` primary variable per `Makefile.am` file. The reason for this is that multiple classes may be generated from a single `.java` file, and the only way to know which classes came from which `.java` file would be to parse the `.java` files. Rather than do this, Automake allows only one JAVA primary per file, so all .class files generated within a given build directory are installed in the location specified by the single JAVA primary variable prefix.

Realizing this gives me pause for thought. It seems that I've broken this rule by assuming in my `java-headers.stamp` rule that the source for class information is the list of files specified in the `dist_noinst_JAVA` variable. In reality, I should probably be looking in the current build directory for all `.class` files found after the rules for the `JAVA` primary are executed.

It's a good thing I don't need to install my JNI header files. I have no way of knowing what they're called from within my `Makefile.am` file! You should by now be able to see the problems that Autotools has with Java. In fact, these problems are not so much related to the poor design of Autotools, as they are the poor design of the Java language itself. This will become clear in the next section, as I cover the rules that build the CSharp native interfaces.

Building the CSharp sources

Returning now to the `xflaim/src/cs` directory brings us to a discussion of building source for a language for which Automake has *no* support: CSharp. Here's the `Makefile.am` file that I wrote for the `cs` directory:

```
SUBDIRS = wrapper

XFLAIM_INCLUDE=-I$(srcdir)/..

noinst_LTLIBRARIES = libxfcsi.la
```

```
libxfcsi_la_SOURCES = \
 Backup.cpp \
 DataVector.cpp \
 Db.cpp \
 DbInfo.cpp \
 DbSystem.cpp \
 DbSystemStats.cpp \
 DOMNode.cpp \
 IStream.cpp \
 OStream.cpp \
 Query.cpp

libxfcsi_la_CPPFLAGS = $(XFLAIM_INCLUDE) $(FTK_INCLUDE)
```

Not surprisingly, this looks almost identical to the `Makefile.am` file found in the `xflaim/src/java` directory. I'm building a simple convenience library from C++ source files found in this directory, just as I did in the `java` directory. As in the `java` version, this makefile is specifying a sub-directory called `wrapper`, which Automake builds first.

The `wrapper/Makefile.am` file looks like this:

```
EXTRA_DIST = xflaim cstest sample xflaim.ndoc

xfcs_sources = \
 xflaim/BackupClient.cs \
 xflaim/Backup.cs \
 ...
 xflaim/RestoreClient.cs \
 xflaim/RestoreStatus.cs
```

```
cstest_sources = \
 cstest/BackupDbTest.cs \
 cstest/CacheTests.cs \
 ...
 cstest/StreamTests.cs \
 cstest/VectorTests.cs

TESTS = cstest_script

AM_CSCFLAGS = -d:mono -nologo -warn:4\
 -warnaserror+ -optimize+
#AM_CSCFLAGS += -debug+ -debug:full\
# -define:FLM_DEBUG

all-local: xflaim_csharp.dll

clean-local:
        -rm xflaim_csharp.dll xflaim_csharp.xml
        -rm cstest_script cstest.exe libxflaim.so
        -rm Output_Stream
        -rm -rf abc backup test.*

check-local: cstest.exe cstest_script

install-exec-local:
        test -z "$(libdir)" || \
```

```
         $(MKDIR_P) "$(DESTDIR)$(libdir)"
        $(INSTALL_PROGRAM) xflaim_csharp.dll\
        "$(DESTDIR)$(libdir)"

install-data-local:
        test -z "$(docdir)" || \
         $(MKDIR_P) "$(DESTDIR)$(docdir)"
        $(INSTALL_DATA) xflaim_csharp.xml\
          "$(DESTDIR)$(docdir)"

uninstall-local:
        rm "$(DESTDIR)$(libdir)/xflaim_csharp.dll"
        rm "$(DESTDIR)$(docdir)/xflaim_csharp.xml"

xflaim_csharp.dll: $(xfcs_sources)
        @list1='$+'; list2=; \
        if test -n "$$list1"; then \
          for p in $$list1; do \
            if test -f $$p; then d=; \
            else d="$(srcdir)/"; fi; \
            list2="$$list2 $$d$$p"; \
          done; \
          echo '$(CSC) -target:library\
           $(AM_CSCFLAGS) $(CSCFLAGS) -out:$@\
           -doc:$(@:.dll=.xml) '"$$list2"; \
          $(CSC) -target:library $(AM_CSCFLAGS)\
```

```
            $(CSCFLAGS) -out:$@ -doc:$(@:.dll=.xml)\
            $$list2; \
        else :; fi

cstest.exe: xflaim_csharp.dll $(cstest_sources)
        @list1='$(cstest_sources)'; \
         list2=; if test -n "$$list1"; then \
          for p in $$list1; do \
            if test -f $$p; then d=; \
            else d="$(srcdir)/"; fi; \
            list2="$$list2 $$d$$p"; \
          done; \
          echo '$(CSC) $(AM_CSCFLAGS) $(CSCFLAGS)\
           -out:$@ '"$$list2"'\
           -reference:xflaim_csharp.dll'; \
          $(CSC) $(AM_CSCFLAGS) $(CSCFLAGS)\
           -out:$@ $$list2\
           -reference:xflaim_csharp.dll; \
        else :; fi

libxflaim.so:
        $(LN_S) ../../.libs/libxflaim.so\
         libxflaim.so

cstest_script: cstest.exe libxflaim.so
        echo "#!/bin/sh" > cstest_script
```

```
          echo "$(CSVM) cstest.exe" >> cstest_script

          chmod 0755 cstest_script
```

The default target for this `Makefile.am` file is, of course, the `all` target. I've hooked the `all` target with my own code by implementing the `all-local` target, which depends on a file named `xflaim_csharp.dll`.

NOTE: This executable file name may be a bit confusing to those who are new to CSharp. In essence, the creators of CSharp (Microsoft) designed the CSharp VM to execute Microsoft native (or almost native) binaries. In porting the CSharp virtual machine to Unix, the Mono team decided against breaking the naming conventions defined by Microsoft, so that Microsoft generated programs could be executed by the Mono CSharp virtual machine implementation. Nevertheless, it still suffers from problems that need to be managed occasionally by name-mapping configuration files.

```
...

xfcs_sources = ...

...

all-local: xflaim_csharp.dll

...

xflaim_csharp.dll: $(xfcs_sources)

        @list1='$+'; list2=; \

        if test -n "$$list1"; then \

          for p in $$list1; do \

            if test -f $$p; then d=; \

            else d="$(srcdir)/"; fi; \

            list2="$$list2 $$d$$p"; \

          done; \

          echo '$(CSC) -target:library\

           $(AM_CSCFLAGS) $(CSCFLAGS) -out:$@\

           -doc:$(@:.dll=.xml) '"$$list2"; \

          $(CSC) -target:library $(AM_CSCFLAGS)\

           $(CSCFLAGS) -out:$@ -doc:$(@:.dll=.xml)\
```

```
            $$list2; \
        else :; fi
...
```

The `xflaim_csharp.dll` binary depends on the list of CSharp source files specified in the `xfcs_sources` variable. I take no credit for the commands in this rule. They're copied from the Automake-generated `java/wrapper/Makefile`, and slightly modified to build CSharp binaries from CSharp source files.

This isn't a lesson in building CSharp sources--the point here is that the default target is automatically built by hooking the `all` target via the `all-local` target.

This `Makefile.am` file also builds a set of unit tests in CSharp that test the CSharp language bindings. Here are the relevant portions of the file:

```
...
cstest_sources = ...

TESTS = cstest_script
...
check-local: cstest.exe cstest_script
...
cstest.exe: xflaim_csharp.dll $(cstest_sources)
        @list1='$(cstest_sources)'; \
         list2=; if test -n "$$list1"; then \
          for p in $$list1; do \
            if test -f $$p; then d=; \
            else d="$(srcdir)/"; fi; \
            list2="$$list2 $$d$$p"; \
          done; \
          echo '$(CSC) $(AM_CSCFLAGS) $(CSCFLAGS)\
           -out:$@ '"$$list2"'\
           -reference:xflaim_csharp.dll'; \
```

```
          $(CSC) $(AM_CSCFLAGS) $(CSCFLAGS)\
           -out:$@ $$list2\
           -reference:xflaim_csharp.dll; \
        else :; fi

libxflaim.so:
        $(LN_S) ../../.libs/libxflaim.so\
         libxflaim.so

cstest_script: cstest.exe libxflaim.so
        echo "#!/bin/sh" > cstest_script
        echo "$(CSVM) cstest.exe" >> cstest_script
        chmod 0755 cstest_script
```

The test sources are built into a CSharp executable named `cstest.exe`. The rules state that `cstest.exe` depends on `xflaim_csharp.dll` and the source files. I again copied the commands from the rule for building `xflaim_csharp.dll`, and modified them for building CSharp programs.

Ultimately, the Automake-generated makefile will attempt to execute the scripts or executables listed in the `TESTS` variable, so the idea here is to ensure that all necessary components get built before these files are executed. The `cstest_script` is a script built for the sole purpose of executing the `cstest.exe` binary in the CSharp virtual machine referenced by the `CSVM` variable. This variable was defined in my `configure` script by the code generated by the `AC_PROG_TRY_CSVM` macro.

The script depends on the executable, and on a link to the `libxflaim.so` file. This file must be present in the current directory, or its location must be specified somehow on the `mono` (`$CSVM`) command line. I chose to simply create a link in the current directory to the location of the actual built library--located up a few directories, and then down into the `xflaim/src/.libs` directory.

Manual installation

Since I'm doing everything myself here, I can't rely on Automake to install files for me. I have to write my own installation rules. Here again are the relevant portions of the makefile:

```
...
install-exec-local:
```

```
        test -z "$(libdir)" || \
         $(MKDIR_P) "$(DESTDIR)$(libdir)"
        $(INSTALL_PROGRAM) xflaim_csharp.dll\
        "$(DESTDIR)$(libdir)"

install-data-local:
        test -z "$(docdir)" || \
         $(MKDIR_P) "$(DESTDIR)$(docdir)"
        $(INSTALL_DATA) xflaim_csharp.xml\
          "$(DESTDIR)$(docdir)"

uninstall-local:
        rm "$(DESTDIR)$(libdir)/xflaim_csharp.dll"
        rm "$(DESTDIR)$(docdir)/xflaim_csharp.xml"
...
```

Note that, as per the rules defined in the GNU Coding Standards, the installation targets do not depend on the binaries they install. I don't want `make install` to build anything. If they haven't been built yet, I'll have to exit out of the root account, back into my own user account and build the binaries with `make all` first.

Cleaning up again

As usual, things must be cleaned up properly. The `clean-local` target handles this nicely for me:

```
...
clean-local:
        -rm xflaim_csharp.dll xflaim_csharp.xml
        -rm cstest_script cstest.exe libxflaim.so
        -rm Output_Stream
        -rm -rf abc backup test.*
```

```
...
```

Configuring compiler options

The original GNU build system was doing a lot for the user. By specifying a list of auxiliary targets on the `make` command line, the user could indicate that she wanted a debug or release build, force a 32-bit build on a 64-bit system, indicate that she wanted to generate generic SPARC code on a Solaris sytem, etc.
Oddly, this turn-key approach to build systems is quite common in commercial code. Whereas, in open source code, the more common practice is to omit much of this framework, allowing the user to set her own options in the standard user variables, `CC`, `CPP`, `CXX`, `CFLAGS`, `CXXFLAGS`, `CPPFLAGS` and others. What's strange about this situation is that commercial software is developed by experts working in the industry, while open source software is often built and consumed by hobbyists. And yet the experts are the ones using the menu-driven rigid-options framework, while the hobbyists have to manually configure their compiler options.
I suppose the most reasonable explanation for this is that commercial software relies on carefully crafted builds that must be able to be duplicated. Open source hobbyists are more carefree, and would rather not give up the flexibility afforded by the lack of such turn-key systems.

To this end, I've added *some* of the options supported by the original GNU makefile-based build system, but left others out. Here's the portion of the `configure.ac` file that I'm talking about:

```
...

# Configure global pre-processor definitions.

AC_DEFINE([_REENTRANT], [],

  [Define for reentrant code])

AC_DEFINE([_LARGEFILE64_SOURCE], [],

  [Define for 64-bit data files])

AC_DEFINE([_LARGEFILE_SOURCE], [],

  [Define for 64-bit data files])



# Configure supported platforms' compiler and li...

case $host in

  sparc-*-solaris*)

    LDFLAGS="$LDFLAGS -R /usr/lib/lwp"
```

```
    if "x$CXX" != "xg++"; then
      if "x$debug" = xno; then
        CXXFLAGS="$CXXFLAGS -xO3"
      fi
      SUN_STUDIO=`"$CXX" -V | grep "Sun C++"`
      if "x$SUN_STUDIO" = "xSun C++"; then
        CXXFLAGS="$CXXFLAGS -errwarn=%all\

-errtags -erroff=hidef,inllargeuse,doubunder"
      fi
    fi ;;

  *-apple-darwin*)
    AC_DEFINE([OSX], [],
      [Define if building on Apple OSX.]) ;;

  *-*-aix*)
    if "x$CXX" != "xg++"; then
      CXXFLAGS="$CXXFLAGS -qthreaded -qstrict"
    fi ;;

  *-*-hpux*)
    if "x$CXX" != "xg++"; then
      # Disable "Placement operator delete
      # invocation is not yet implemented" warning
      CXXFLAGS="$CXXFLAGS +W930"
```

```
    fi ;;

esac

...
```

Here, I've used the `$host` variable to determine the type of system for which I'm building. The `config.guess` and `config.sub` files are your friends here. If you need to write code like for your project, then you'll need to examine these files to find common traits for the processes and systems for which you'd like to set various compiler and linker options. Note also that in each of these cases (except for the definition of the `OSX` preprocessor variable on Apple Darwin systems), I'm really only setting flags for native compilers. The GNU compiler tools seem to be able to handle any sort of code thrown at them without monkeying around with compiler options. This is a good thing, and a lesson could be learned by compiler vendors from this fact.

Hooking Doxygen into the build process

I wanted to generate documentation as part of my build process, if possible. That is, if the user has doxygen installed on her system, then the build system will use it to build doxygen documentation as part of the `make all` process. As I've already mentioned, I used the `AM_CONDITIONAL` macro to conditionally build the docs directory.
Now, relative to the xflaim project, this is probably not the right thing to do, as I want non-doxygen documentation to be installed even if doxygen *isn't* available. The right approach to this problem would be to have a `doxygen` directory beneath the `docs` directory that handles only generated documentation. The `docs` directory itself would be limited to simply installing existing documentation. I've combined them to save space in this book, but I'll probably fix this problem before committing my build system to the project.
For the FLAIM tool kit project, this configuration works fine for now, because there is no other documentation to be installed. I say "for now" because at some point in the future, someone may write some tool kit documentation, and then I'll have to move things around to get the end-user experience I want.

Doxygen uses a configuration file (often called `doxyfile`) to configure literally hundreds of doxygen options. This configuration file contains some information that is known to Autoconf. This sounds like the perfect opportunity to use an Autoconf-generated file. To this end, I've written a file called `doxyfile.in` that contains most of what a normal doxyfile would contain, except it also has a few Autoconf substitution variable references:

```
...

PROJECT_NAME           = @PACKAGE_NAME@

PROJECT_NUMBER         = @PACKAGE_VERSION@

...

STRIP_FROM_PATH        = @top_srcdir@

...
```

There are many other lines in this file, but they are all identical to the output file, so I've omitted them for the sake of space and clarity. The key here is that Autoconf will replace these values with those defined in `configure.ac`, and by Autoconf itself. If these values change in `configure.ac`, the generated file will be written with the new values.

I've added a reference to `ftk/docs/doxyfile` to the `AC_CONFIG_FILES` list in ftk's `configure.ac` file. That's all it takes.

Here's the `ftk/docs/Makefile.am` file:

```
docpkg = $(PACKAGE_TARNAME)-doxy-$(PACKAGE_VERSION).tar.gz

doc_DATA = $(docpkg)

$(docpkg): doxygen.stamp
        tar chof - html | gzip -9 -c >$@

doxygen.stamp: doxyfile
        $(DOXYGEN) $(DOXYFLAGS) $<
        echo Timestamp > $@

CLEANFILES = doxywarn.txt doxygen.stamp $(docpkg)

clean-local:
        -rm -rf html
```

In this file, I've created a package name for the tarball that will contain the doxygen documentation files. It's basically the same as the distribution tarball for the ftk project, except that it contains the text `-doxy` after the package name.

I've also defined a `doc_DATA` variable containing the name of the doxygen tarball. This file will be installed in the `$(docdir)` directory, which by default is `$(datarootdir)/doc/$PACKAGE_TARNAME`. And `$(datarootdir)` is configured as `$(prefix)/share`, by default.

Note again here that the `DATA` primary brings with it significant Automake functionality--installation is managed automatically. And, while I must build the doxygen documentation package myself, the `DATA` primary automatically hooks the `all` target for me, so that my package is built when the user executes `make all`.

I'm using another stamp file here because doxygen generates literally hundreds of html files from my input file (and from the source tree). Rather then attempt to figure out a rational way to assign dependencies, I simply generate one stamp file, and then use that to determine whether or not the documentation is out of date.

Note that this is wrong, but much simpler than attempting to list every source file used in the generation of the documentation as a dependency of the stamp file. (*In fact, this is quite trivial in this project because the only source file currently containing documentation markup, and thus, listed in the doxyfile as an input file, is the `flaimtk.h` header file. However, this could easily change in the future.*)

For cleaning my generated files, I've used a combination of the `CLEANFILES` variable and a clean-local rule--just to show you that it can be done.

Adding a new `rpms` target

Adding a new non-standard target is a little different than hooking an existing target. In the first place, you don't need to use `AM_CONDITIONAL` and Autoconf checks to see if you have the tools you need. You may do everything from the `Makefile.am` file, if you wish. After all, if the user was building on a Debian system, why in the world did she type `make rpms` in the first place?! Nonetheless, you still have to account for the possibility that the user will experiment.

First, I created a directory called `obs` to contain the `Makefile.am` file for building RPM package files. OBS is an acronym for "Opensuse Build Service", which is an online package building service (found at http://build.opensuse.org) that I fell in love with almost as soon as it came out. I've had some experience building distro packages, and I can tell you, it's far less painful with the OBS than it is using more traditional techniques. Furthermore, packages built with the OBS can be published on the OBS web site for others to access immediately after they're built (in this case, http://software.opensuse.org/search).

Building RPM package files is done using a configuration file, called a "spec" file, which is very much like the doxyfile is used to configure doxygen for a specific project. As with the doxyfile, the rpm spec file contains information that Autoconf knows about regarding the project package. So, I wrote an `ftk.spec.in` file, adding substitution variables where appropriate, and then I added another file reference to the `AC_CONFIG_FILES` macro. Here is the relevant portion of the `ftk.spec.in` file:

```
Name: @PACKAGE_TARNAME@

BuildRequires: gcc-c++ libstdc++ libstdc++-devel doxygen

Summary: FTK is the FLAIM cross-platfomr toolkit.

URL: http://forge.novell.com/modules/xfmod/project/?flaim

Version: @PACKAGE_VERSION@

Release: 1

License: GPL

Vendor: Novell, Inc.

Group: Development/Libraries/C and C++

Source: %{name}-%{version}.tar.gz

BuildRoot: %{_tmppath}/%{name}-%{version}-build
```

```
...
```

I used `@PACKAGE_TARNAME@` and `@PACKAGE_VERSION@`. Now the tar name is not likely to change much over the life time of this project, but the version will change quite often. Without the Autoconf substitution mechanism, I'd have to remember to update this version number whenever I updated the version in the `configure.ac` file. Here's the `obs/Makefile.am` file:

```
rpmspec = $(PACKAGE_TARNAME).spec

rpmmacros =\
 --define='_rpmdir $(PWD)'\
 --define='_srcrpmdir $(PWD)'\
 --define='_sourcedir $(PWD)'\
 --define='_specdir $(PWD)'\
 --define='_builddir $(PWD)'

rpmopts = --nodeps --buildroot='$(PWD)/_rpm'

rpmcheck:
	@which rpmbuild &> /dev/null; \
	if [ $$? -ne 0 ]; then \
	  echo "*** This make target requires an rpm-based linux distribution."; \
	  (exit 1); exit 1; \
	fi

srcrpm: rpmcheck $(rpmspec)
	rpmbuild -bs $(rpmmacros) $(rpmopts) $(rpmspec)
```

```
rpms: rpmcheck $(rpmspec)

        rpmbuild -ba $(rpmmacros) $(rpmopts) $(rpmspec)



.PHONY: rpmcheck srcrpm rpms
```

Building RPM package files is rather simple, as you can see. The targets provided by this makefile include `srcrpm` and `rpms`. The `rpmcheck` target is only used internally. How can you tell? Well, you can't really tell from here. In order to find out which targets in a lower-level `Makefile.am` file are supported by a top-level build, you have to look at the top-level `Makefile.am` file:

```


...

rpms srcrpm: dist

        $(MAKE) -C obs $(AM_MAKEFLAGS) $@

        rpmarch=`rpm --showrc | grep ^build\ arch | sed 's/\(.*:
\)\(.*\)/\2/'`; \

        test -z $$rpmarch || ( mv $$rpmarch/* .; rm -rf
$$rpmarch )

        -rm -rf $(distdir)

...

.PHONY: srcrpm rpms
```

As you can see from the first command in this rule, when a user targets `rpms` or `srcrpm` from the top-level build directory, the commands are recursively passed on to the `obs/Makefile`. The remaining commands simply remove droppings left behind by the RPM build process that are simpler to remove at this level. (Try building an rpm sometime, and you'll see what I mean!)

Notice also that both of these top-level makefile targets depend on the `dist` target. That's because the RPM build process requires the distribution tarball. Adding it as a dependency simply ensures that the distribution tarball is there when the `rpmbuild` utility needs it.

Summary

While using Autotools, there are a myriad of details to manage, most of which, as they say in the free software world, "can wait for the next release!" The take-away lesson here is that a build system is never really finished. It should be incrementally improved over time, as you find time in your schedule to work on it. And it can be rewarding to do so.

I've shown you a number of new features--features I didn't cover directly in the earlier chapters on the individual tools. There are many many more features that I couldn't begin to cover. You'll need to study the GNU Autotools manuals to become truly proficient. At this point, it should be pretty simple to pick up this additional information yourself.

Source Code

You can access the entire flaim project source hierarchy, along with the new build system defined in this chapter from the attached source archive.

‹ Chapter 5: Building shared libraries with Libtool up Chapter 7: A catalog of reusable solutions ›

Chapter 7: A catalog of reusable solutions

Mon, 2008-03-10 20:35 -- John Calcote

This chapter started out as a catalog of reusable solutions--canned macros, if you will. But as I finished chapter after chapter preceeding this one, it became clear to me that I really needed to broaden my definition of a "canned solution". Instead of just cataloging interesting macros here (which has been done before anyway), this chapter lists several unrelated, but important tips for creating great projects in general. Some of these are related to the GNU Autotools, but others are merely good programming practice with respect to open source and free software projects.

Never expose `config.h` in a public interface

At times, I've come across poorly designed library interfaces where a project's `config.h` file is required by the project's public header files. This presents a real problem when more than one such library is required by a consumer. Which `config.h` file should be included? Both are named the same, and chances are good that both provide similar--even identically named--definitions.

When you carefully consider the original purpose of `config.h`, then you can see that it makes little sense to expose it in a library's public interface (by including it in any of the library's public header files). Its purpose is to provide platform-specific definitions to a particular build of the library. On the other hand, the public interface of a portable library is, by definition, platform-independent.

Interface design is a fairly general topic in computer science. This item focuses a bit more specifically on designing great Application Programmer Interfaces (API's) for GNU Autotools library projects. Specifically, how to avoid including `config.h` in your public interfaces.

But this item provides some more or less generic advice, as well. When designing a library for consumption by other projects, you have a great responsibility to not polute your consumers' symbol spaces with useless garbage from your header files. I once worked on a project that consumed a library interface from another team. This team provided both a Win32 and a Unix version of their library, with the header file being "portable" between the two platforms. Unfortunately, they didn't understand the definition of a clean interface. At some point in their public header files, they had a bit of code that looked like this:

```
#ifdef _WIN32

# include <windows.h>

#else

# typedef void * HANDLE

#endif
```

Ouch! Did they really need to include `windows.h`--just for the definition of `HANDLE`? Not only should they not have done this, but in fact, they probably should have used a different name for the handle object in their public interface. Why? Because `HANDLE` is too generic, and could easily conflict with a dozen other library interfaces. Why not use `XYZ_HANDLE`, or something a little more specific to the `XYZ` library?

In C++ this concept is even simpler to implement with the use of namespaces. Anyone who properly understands the rationale behind C++ namespaces will have no problem understanding the value of this advice.

To properly design a library, first design the public interface such that it exposes as little of the internals of your library implementation as is reasonable. Now, *you'll* have to determine the definition of reasonable, but it will most probably involve a compromise between abstraction and performance.

When designing an API, start with the functionality you wish to expose from your library. Design functions that will maximize ease of use for your consumers. If you find yourself trying to decide between a simpler implementation and a simpler user experience, always err on the side of ease of use for your consumers. They'll thank you by actually *using* your library. Of course, if the interface is already defined by a software standard, then much of your work is done for you, but often this is not the case and you will have to make these decisions.

Next, try to abstract away internal details. Everyone knows that the C language doesn't make it very easy to do this. You often need to pass structure references in public API's which contain internal details of your implementation that consumers have no business seeing. Ironically, C++ is just as bad in this area. C++ classes define public interfaces and private implementation details in the same class definition.

In C, a common solution for this problem is to define a public alias for a private structure in terms of a `void` pointer. Many developers don't care for this approach because it reduces type safety in the interface. Such losses of type safety occur often in C programming. It's the nature of the language. The loss of type safety is significantly offset by the increase in interface abstraction. Here's an example of this technique:

Private C source file

```
#include <abc_pub.h>


#if HAVE_CONFIG_H
# include <config.h>
#endif


typedef struct {
   /* private details */
} abc_impl;


int abc_func(abc * p)
```

```
{
   abc_impl * ip = (abc_impl *)p;
   /* use 'p' through 'ip' */
}
```

Public C header file - `abc_pub.h`

```
typedef void abc;
int abc_func(abc * p);
```

Notice how the abstraction so conveniently alleviates the need to include a bunch of really private definitions in the library's public interface.

In C++, this can be done using a few different techniques, including virtual interfaces, and the PIMPL (Private IMPLementation) pattern.

In the PIMPL pattern, implementation details are hidden behind a pointer to a private implementation class stored as private data within the public interface class. Here's an example of the PIMPL pattern:

Private C++ source file

```
#include <abc_pub.h>

#if HAVE_CONFIG_H
# include <config.h>
#endif

class abc_impl {
   /* private details */
};

int abc::func(void)
```

```
{
   /* use 'pimpl' pointer */
}
```

Public C++ header file - `abc_pub.h`

```
class abc_impl;
class abc {
  abc_impl * pimpl;
public:
  int func(void);
};
```

The C++ language allows the use of a forward declaration for any types used only through references or pointers, but never dereferenced in the public interface. Thus, the definition of the implementation class need not be exposed in the public interface, because the compiler is quite happy to compile the public interface files without the definition of the private implementation class.

The performance trade-off here generally involves the dynamic allocation of an instance of the private implementation class, and then accessing class data indirectly through this pointer, rather than directly in the public structure. Again, however, notice how all internal details are now conveniently hidden, and thus not required by the public interface.

Another approach when using C++ is to define a public "interface" class, most (if not all) of whose methods are declared *pure virtual*. The interface is then implemented internally by the library. To access an object of this class, consumers call a public *factory* function, whose job it is to return a pointer to the implementation class in terms of the interface definition:

Private C++ source file

```
#include <abc_pub.h>

#if HAVE_CONFIG_H
# include <config.h>
#endif
```

```
class abc_impl : public abc {

   /* implementation of virtual methods */

};
```

Public C++ header file - `abc_pub.h`

```
#define interface class



interface abc {

public:

  virtual int func(void) = 0;

};



abc * abc_instantiate(/* abc_impl ctor params */);
```

To show the policy in practice here, I've used the C++ preprocessor to define a new keyword, *interface*. By definition, interface is synonymous with class, so they may be used interchangably. The idea here is that an interface doesn't expose any implementation details to the consumer. The public library function *abc_instantiate* returns a pointer to a new object of type abc_impl, except in terms of *abc*. Thus, nothing internal need be shown to the caller in the public header file.
You may think the interface class method is more efficient than the PIMPL method, but the fact is most compilers implement virtual function calls as tables of function pointers referred to by a hidden "vptr" address within the implementation class, so you still end up calling all of your public methods indirectly through a pointer. Which of these techniques you choose to use to help you hide your implementation details is more a matter of taste than performance.

When I design a library, I start by designing a minimal but complete functional interface, with as much of my internal implementation abstracted away as is reasonable. I try to use only standard library basic types, if possible, in my function prototypes, and then include only the C or C++ standard header files required by the use of those types and definitions. This technique is the fastest way I've found to creating a highly portable and maintainable interface.

If you still can't see the value in the advice offered by this item, then let me give you one more scenario to ponder. Consider what happens when a Linux distro packager decides to create a 'devel' package for your library - that is, a package containing static libraries and header files, designed to be installed into the `/user/lib` and `/usr/include` directories on a target system. Every header file required by your library must be installed into the `/user/include` directory. If your library's public interface requires the inclusion of your `config.h` file, then by extension, your `config.h` must be installed into the

`/usr/include` directory. Now consider what happens when multiple such libraries need to be installed.
I've seen message threads on the Autotools mailing list defending the need to do this sort of thing, and providing techniques for naming `config.h` in a package-specific manner. These techniques often involve some form of post-processing of this file to rename the macros it contains such that they don't conflict with other packages' installed `config.h` macros. While this can be done, and while there are a few good reasons for doing so (usually involving a legacy code base that can't be modified much), these situations should be considered the exception, not the rule.

Implementing recursive extension targets

When you add a new top-level target to your build system, you have to either tie it into an existing Automake target, or add your own make code to the target to recurse into the sub-directory structure provided by Automake in your build system.

The SUBDIRS variable can be used to recurse all sub-directories of the current directory, passing the build command into the makefiles in these directories. This works great for targets that must be built based on configuration options, because after configuration the `SUBDIRS` variable contains only those directories configured to be built.
If you need to execute your new recursive target in all sub-directories, regardless of any conditional configuration, which might exclude one or more sub-directories specified in the `SUBDIRS` variable, then use the `DISTDIRS` variable instead. This variable is derived by Automake from all conditional and non-conditional additions to the `SUBDIRS` variable. There are various ways to recurse, including some really simple one-liners provided by GNU Make specific syntax. but the most portable way is to use the technique that Automake itself uses:

```
recursive-target:

  $preorder_commands

  for dir in $(SUBDIRS); do \

    (cd $$dir && $(MAKE) $(AM_MAKEFLAGS) $@) || exit 1; \

  done

  $postorder_commands



.PHONY: recursive-target
```

The `$preorder_commands` macro can be used to do things that must be done before recursing to lower-level directories. The `$postorder_commands` macro can likewise be used to do additional things once you return from the lower-level directories. At some point in the hierarchy, you'll need to actually do something useful besides calling down to lower levels. Use these two macros to encode the actual functionality of this technique. For example, assuming you want to build some generated documentation, you might have a special target called `doxygen`. Even if you happen to be okay with building your documentation in the top-level directory, there may be cases where you need to distribute the generation of your documentation to various directories within your project hierarchy. You might use the following code in each `Makefile.am` file in your project:

```
# uncomment if doxyfile exists in this directory
# postorder_commands = $(DOXYGEN) $(DOXYFLAGS) doxyfile

doxygen:
  $preorder_commands
  for dir in $(SUBDIRS); do \
    (cd $$dir && $(MAKE) $(AM_MAKEFLAGS) $@) || exit 1; \
  done
  $postorder_commands

.PHONY: doxygen
```

For directories where `doxyfile` doesn't exist, you may comment out (or better yet, simply omit) the `postorder_commands` macro definition. The `doxygen` target will be harmlessly propagated to the next lower level in the build tree.
This code ensures that the build terminates when a lower-level makefile fails on the recursive target, propagating the shell error code (1) back up to each parent until the top-level shell is reached. This is important, or the build may continue at some levels until a different error is encountered.

Also note that I don't use the somewhat less portable `-C` make command line option to change directories before running the sub-make operation.
Allow me to emphasize here that if you choose to implement a completely recursive global target in this manner, then you must include this code snippet in every single Makefile.am file in your project, even if it has nothing to do with the generation of documentation. If you don't, then make will fail on that makefile because no such target exists within that makefile. The commands may do nothing, but the target must exist.

If you want to do something simpler, such as pass the target down to a single sub-directory beneath the top-level directory--say, a `doc` directory just below the top--then life is simpler:

Top-level `Makefile.am`

```
doxygen:
  (cd doc && $(MAKE) $(AM_MAKEFLAGS) $@) || exit 1
```

```
.PHONY: doxygen
```

doc directory Makefile.am

```
doxygen:

	$(DOXYGEN) $(DOXYFLAGS) doxyfile



.PHONY: doxygen
```

NOTE: The variables, `DOXYGEN` and `DOXYFLAGS` are assumed to exist by virtue of some macro or shell code executed within the configure script.

Using a repository revision number

Arguments to the Autoconf `AC_INIT` macro must be static text. That is, they can't be shell variables, and Autoconf will flag attempts to use shell variables in these arguments as errors. This is all well and good until you want to calculate any portion of your package's version number during the configuration process.

I once tried to use a shell variable in the `VERSION` argument so that I could substitute my Subversion revision number into the `VERSION` argument when configure was executed. I spent a couple of days trying to figure out how to trick Autoconf into letting me use a shell variable as a sort of "revision" field in my package's version number. Eventually, I discovered the following trick, which I implemented in my `configure.ac` script, and in my top-level `Makefile.am` file:

configure.ac

```
SVNREV=`svnversion $srcdir | sed 's/:.*//'`

which svnversion > /dev/null; \

if [ $? -ne 0 ] || [ "x$SVNREV" = "xexported" ]

	then SVNREV=`cat $srcdir/SVNREV`

	else echo -n $SVNREV>$srcdir/SVNREV

fi

AC_SUBST(SVNREV)
```

First, the shell variable `SVNREV` is set to the output of the `svnversion` command, as executed on the project top-level source directory. The output is piped through the `sed` utility to remove all text following an embedded COLON (:) character. This gives us a raw Subversion revision number--that is, *if* the code is executed in a true Subversion work area, which isn't always the case.

When a user executes this `configure` script from a distribution tarball, Subversion may not even be installed on her workstation. Even if it is, the top-level project directory comes from the tarball, not a Subversion repository. To handle these situations, the next line checks to see if either Subversion is not installed, or if the output from the first line was the word, "exported", which is the result of executing the `svnversion` utility on a non-work-area directory.

If either of these cases is true, then the `SVNREV` variable is populated from the contents of a file called `SVNREV`. This file actually ships with a distribution tarball containing this configuration code. This is true because if the `svnversion` command works properly, generating a true Subversion repository revision number, then that value is immediately written to the `SVNREV` file by the `else` clause of this `if` statement.

Finally, `AC_SUBST` is used to cause Autoconf to substitute the `SVNREV` variable so that it becomes available to the makefile as a `make` variable (all `AC_SUBST` variables are converted to `make` variables by Automake).

In the top-level `Makefile.am` file, I then ensure that the `SVNREV` file becomes part of the distribution tarball by adding it to the `EXTRA_DIST` list. This means that when a distribution tarball is created and published by the maintainer, it contains an `SVNREV` file that contains the source tree revision number to be used when generating a tarball from this source code. It's accurate because the tarball was actually generated from a this revision of the SVN repository.

Generally, it's not particularly important that a tarball be able to generate a proper tarball, but an Automake-generated tarball can do so without this code, so it should be able to do so *with* this code.

Top-level `Makefile.am`

```
EXTRA_DIST = SVNREV

distdir = $(PACKAGE)-$(VERSION).$(SVNREV)
```

The `distdir` make variable controls the name of the distribution directory and the tarball file name generated by Automake. Setting this variable in the top-level `Makefile.am` file affects the generation of the distribution tarball, because the top-level `Makefile.am` is where this functionality is located.

If you have a particular need for the `distdir` variable to be formatted correctly in any other `Makefile.am` file in your project, you should set this variable in that file as well. For most purposes, setting it in the top-level `Makefile.am` file should be sufficient.

Ensure your distribution packages are "clean"

Have you ever downloaded and unpacked an open source package, and tried to run `configure; make` only to have it fail half way through one of these steps? As you dug into the problem, you perhaps discovered that there were missing files in the tarball. How sad to have this happen on an Autotools project, when the Autotools make it so easy to ensure that this simply doesn't happen.

Ensure that your distribution tarballs are always clean and complete by running the `distcheck` target on a newly created tarball. Don't be satisified with what you *believe* about your package. Allow Automake to run the distribution unit tests, so to speak. I call these tests "unit tests" because they provide the same testing functionality for a distribution package that regular unit tests provide for your code.

You'd never make a code change and ship a package without running your unit tests, would you? (If so, then you can safely skip this section.) So don't ship your tarballs without running the build system unit tests either - run `make distcheck` on your project *before* posting your new tarballs.

Cross-compilation

Emulating autoconf replacement techniques

Using the ac-archive project

...MORE TO COME...

‹ Chapter 6: FLAIM: an Autotools example up Appendix A: An overview of the M4 macro processor ›

References

Mon, 2008-03-10 20:38 -- John Calcote

The GNU Autoconf Manual
The GNU Automake Manual
The GNU Libtool Manual
GNU Autoconf, Automake and Libtool by Gary V. Vaughan, Ben Elliston, Tom Tromey and Ian Lance Taylor. Also known as "The Goat Book".
An Autotools Tutorial by A. Duret-Lutz. This is a great tutorial, but one of the nicest aspects of this web site is the list of Autotools resources at the bottom of the first page.
The GNU M4 Macro Processor Manual
The GNU Make Utility Manual
The GNU Coding Standards
The GNU sed Manual
Bruce Barnett's very well-written Unix utility tutorials
The AC Autoconf Macro Archive - an archive of drop-in components for autoconfiscated projects. Macros are fully classified for easy lookup, and full documentation for each macro is provided.