CHAPTER 45

Compiling Tcl and Extensions

This chapter explains how to build Tcl from the source distribution, and how to create C extensions that are built according to the standard Tcl Extension Architecture (TEA).

Compiling Tcl from the source distribution is easy. One of the strengths of Tcl is that it is quite portable, so it has been built on all kinds of systems including Unix, Windows, Macintosh, AS/400, IBM mainframes, and embedded systems. However, it can be a challenge to create a Tcl extension that has the same portability. The Tcl Extension Architecture (TEA) provides guidelines and samples to help extension authors create portable Tcl extensions. The TEA is a result of collaboration within the Tcl user community, and it will continue to evolve.

This chapter starts with a walk through of how Tcl itself is built. This serves as a model for building extensions. There are also some by-products of the Tcl build process that are designed to make it easier to build your extensions. So if you are an extension author, you will almost always want to get started by compiling Tcl itself.

You can find the Tcl and Tk sources on the CD-ROM, and on the Web:

```
http://www.scriptics.com/products/tcltk/
```

Source distributions can be found at the Scriptics FTP site:

```
ftp://ftp.scriptics.com/pub/tcl/
```

The on-line CVS repository for Tcl software is explained here:

```
http://www.scriptics.com/products/tcltk/netcvs.html
```

If you have trouble with these URLs, please check this book's Web site for current information about the Tcl sources:

```
http://www.beedub.com/book/
```

Standard Directory Structure

The Source Distribution

Table 45–1 describes the directory structure of the Tcl source distribution. The Tk distribution is similar, and you should model your own source distribution after this. It is also standard to place the Tcl, Tk, and other source packages under a common source directory (e.g., `/usr/local/src` or `/home/welch/cvs`). In fact, this may be necessary if the packages depend on each other.

Table 45–1 The Tcl source directory structure.

`tcl8.2`	The root of the Tcl sources. This contains a `README` and `license_terms` file, and several subdirectories.
`tcl8.2/compat`	This contains `.c` files that implement procedures that are otherwise broken in the standard C library on some platforms. They are only used if necessary.
`tcl8.2/doc`	This contains the reference documentation. Currently this is in *nroff* format suitable for use with the UNIX *man* program. The goal is to convert this to XML.
`tcl8.2/generic`	This contains the generic `.c` and `.h` source files that are shared among Unix, Windows, and Macintosh.
`tcl8.2/mac`	This contains the `.c` and `.h` source files that are specific to Macintosh. It also contains *Code Warrior* project files.
`tcl8.2/library`	This contains init.tcl and other Tcl files in the standard Tcl script library.
`tcl8.2/library/encoding`	This contains the Unicode conversion tables.
`tcl8.2/library/`*`package`*	There are several subdirectories (e.g., `http2.0`) that contain Tcl script packages.
`tcl8.2/test`	This contains the Tcl test suite. These are Tcl scripts that exercise the Tcl implementation.
`tcl8.2/tools`	This is a collection of scripts used to help build the Tcl distribution.
`tcl8.2/unix`	This contains the `.c` and `.h` source files that are specific to UNIX. This also contains the *configure* script and the `Makefile.in` template.
`tcl8.2/unix/dltest`	This contains test files for dynamic loading.
`tcl8.2/unix/`*`platform`*	These can be used to build Tcl for several different platforms. You create the *`package`* directories yourself.
`tcl8.2/win`	This contains the `.c` and `.h` source files that are specific to Windows. This also contains the *configure* script and the `Makefile.in` template. This may contain a `makefile.vc` that is compatible with *nmake*.

The Installation Directory Structure

When you install Tcl, the files end up in a different arrangement than the one in the source distribution does. The standard installation directory is organized so that it can be shared by computers with different machine types (e.g., Windows, Linux, and Solaris). The Tcl scripts, include files, and documentation are all in shared directories. The applications and programming libraries (i.e., DLLs) are in platform-specific directories. You can choose where these two groups of files are installed with the `--prefix` and `--exec-prefix` options to *configure*, which is explained in detail in the next section. Table 45–2 shows the standard installation directory structure:

Table 45–2 The installation directory structure.

`arch``/bin`	This contains platform-specific applications. On Windows, this also contains binary libraries (i.e., DLLs). Typical *`arch`* names are `solaris-sparc`, `linux-ix86`, and `win-ix86`.
`arch``/lib`	This contains platform-specific binary libraries on UNIX systems (e.g., `libtcl8.2.so`)
`bin`	This contains platform-independent applications (e.g., Tcl script applications).
`doc`	This contains documentation.
`include`	This contains public `.h` files.
`lib`	This contains subdirectories for platform-independent script packages. Packages stored here are found automatically by the Tcl auto loading mechanism described in Chapter 12.
`lib/tcl8.2`	This contains the contents of the `tcl8.2/library` source directory, including subdirectories.
`lib/`*`package`*	This contains Tcl scripts for *`package`*. Example *`package`* directories include `tk8.2` and `itcl3.0.1`.
`man`	This contains reference documentation in UNIX *man* format.

If you are an expert in *configure*, you may be aware of other options that give you even finer control over where the different parts of the installation go. However, because of the way Tcl automatically searches for scripts and binary libraries, you should avoid deviating from the recommended structure.

Building Tcl from Source

Compiling Tcl from the source distribution is a two-step process: configuration, which uses a *configure* script; then compiling, which is controlled by the *make* program. The *configure* script examines the current system and makes various settings that are used during compilation. When you run *configure*, you make some basic choices about how you will compile Tcl, such as whether you will compile with debugging systems, or whether you will turn on threading support. You also define the Tcl installation directory with *configure*. You use *make* to compile the source code, install the compiled application, run the test suite, and clean up after yourself.

The *make* facility is familiar to any Unix programmer. By using the freely available Cygwin tools, you can use *configure* and *make* on Windows, too. Tcl still uses the Microsoft VC++ compiler; it does not use *gcc* on Windows.

Windows and Macintosh programmers may not have experience with *make*. The source distributions may also include project files for the Microsoft *Visual C++* compiler and the Macintosh *Code Warrior* compiler, and it may be easier for you to use these project files, especially on the Macintosh. Look in the `win` and `mac` subdirectories of the source distribution for these project files. However, the focus of this chapter is on using *configure* and *make* to build your Tcl applications and extensions.

Configure and Autoconf

If you are the developer of a Tcl extension, there is a preliminary setup step that comes before configuration. In this step, you create templates for the *configure* script and the `Makefile` that drives *make*. Then, you use the *autoconf* program to create the *configure* script. By using *autoconf*, a developer on Windows or Linux can generate a *configure* script that is usable by other developers on Solaris, HP-UX, FreeBSD, AIX, or any system that is vaguely UNIX-like.

The three steps: setup, configuration and make, are illustrated by the build process for Tcl and Tk:

- The developer of a source code package creates a `configure.in` template that expresses the system dependencies of the source code. They use the *autoconf* program to process this template into a `configure` script. The developer also creates a `Makefile.in` template. Creating these templates is described later. The Tcl and Tk source distributions already contain the `configure` script, which can be found in the `unix` and `win` subdirectories. However, if you get the Tcl sources from the network CVS repository, you must run *autoconf* yourself to generate the configure script.
- A user of a source code package runs `configure` on the computer system they will use to compile the sources. This step converts `Makefile.in` to a `Makefile` suitable for the platform and configuration settings. If you have only one platform, simply run *configure* in the `unix` (or `win`) directory:

  ```
  % cd /usr/local/src/tcl8.2/unix
  % ./configure flags
  ```

  The *configure* flags are described in Table 45–3. I use `./configure` because I do not have `.` on my `PATH`. Furthermore, I want to ensure that I run the configure script from the current directory! If you build for multiple platforms, create subdirectories of `unix` and run *configure* from there. For example, here we use `../configure`:

```
% cd /usr/local/src/tcl8.2/unix
% mkdir solaris
% cd solaris
% ../configure flags
```

- The *configure* script uses the `Makefile.in` template to generate the `Makefile`. Once *configure* is complete, you build your program with *make*:

  ```
  % make
  ```

  You can do other things with *make*. To run the test suite, do:

  ```
  % make test
  ```

  To install the compiled program or extension, do:

  ```
  % make install
  ```

The Tcl Extension Architecture defines a standard set of actions, or make targets, for building Tcl sources. Table 45–4 on page 650 shows the standard *make* targets.

Make sure you have a working compiler.

As the configure script executes, it prints out messages about the properties of the current platform. You can tell if you are in trouble if the output contains either of these messages:

```
checking for cross compiler ... yes
```

or

```
checking if compiler works ... no
```

Either of these means that *configure* has failed to find a working compiler. In the first case, it assumes that you are configuring on the target system but will cross-compile from a different system. *Configure* proceeds bravely ahead, but the resulting Makefile is useless. While cross-compiling is common on embedded processors, it is rarely necessary on UNIX and Windows. I see this message only when my UNIX environment isn't set up right to find the compiler.

Many UNIX venders no longer bundle a working compiler. Fortunately, the freely available *gcc* compiler has been ported to nearly every UNIX system. You should be able to search the Internet and find a ready-to-use *gcc* package for your platform.

On Windows there is a more explicit compiler check, and *configure* exits if it cannot find the compiler. Tcl is built with the Microsoft *Visual C++* compiler. It ships with a batch file, `vcvars32.bat`, that sets up the environment so that you can run the compiler from the command line. You should read that file and configure your environment so that you do not have to remember to run the batch file all the time. At this time, other Windows compilers are not supported by the *autoconf* macros.

Standard Configure Flags

Table 45–3 shows the standard options for Tcl configure scripts. These are implemented by a configure library file (`aclocal.m4`) that you can use in your own configure scripts. The facilities provided by `aclocal.m4` are described in more detail later.

Table 45–3 Standard configure flags.

`--prefix=`*`dir`*	Defines the root of the installation directory hierarchy. The default is `/usr/local`.
`--exec-prefix=`*`dir`*	This defines the root of the installation area for platform-specific files. This defaults to the `--prefix` value. An example setting is `/usr/local/solaris-sparc.`
`--enable-gcc`	Uses the *gcc* compiler instead of the default system compiler.
`--disable-shared`	Disables generation of shared libraries and Tcl shells that dynamically link against them. Statically linked shells and static archives are built instead.
`--enable-symbols`	Compiles with debugging symbols.
`--enable-threads`	Compiles with thread support turned on.
`--with-tcl=`*`dir`*	Specifies the location of the build directory for Tcl.
`--with-tk=`*`dir`*	Specifies the location of the build directory for Tk.
`--with-tclinclude=`*`dir`*	Specifies the directory that contains `tcl.h`.
`--with-tcllib=dir`	Specifies the directory that contains the Tcl binary library (e.g., `libtclstubs.a`).
`--with-x11include=dir`	Specifies the directory that contains `X11.h`.
`--with-x11lib=dir`	Specifies the directory that contains the X11 binary library (e.g., `libX11.6.0.so`).

Any flag with `disable` or `enable` in its name can be inverted. Table 45–3 lists the nondefault setting, however, so you can just leave the flag out to turn it off. For example, when building Tcl on Solaris with the *gcc* compiler, shared libraries, debugging symbols, and threading support turned on, use this command:

```
configure --prefix=/home/welch/install \
    --exec-prefix=/home/welch/install/solaris \
    --enable-gcc --enable-threads --enable-symbols
```

Keep all your sources next to the Tcl sources.

Your builds will go the most smoothly if you organize all your sources under a common directory. In this case, you can specify the same configure flags for Tcl and all the other extensions you will compile. In particular, you must use the same `--prefix` and `--exec-prefix` so that everything gets installed together.

If your source tree is not adjacent to the Tcl source tree, then you must use `--with-tclinclude` or `--with-tcllib` so that the header files and runtime library can be found during compilation. Typically, this can happen if you build an extension under your home directory, but you are using a copy of Tcl that has been installed by your system administrator. The `--with-x11include` and `--with-x11lib` flags are similar options necessary when building Tk if your X11 installation is in a nonstandard location.

Installation

The `--prefix` flag specifies the main directory (e.g., `/home/welch/install`). The directories listed in Table 45–2 are created under this directory. If you do not specify `--exec-prefix`, then the platform-specific binary files are mixed into the main `bin` and `lib` directories. For example, the *tclsh8.2* program and `libtcl8.2.so` shared library will be installed in:

```
/home/welch/install/bin/tclsh8.2
/home/welch/install/lib/libtclsh8.2.so
```

The script libraries and manual pages will be installed in:

```
/home/welch/install/lib/tcl8.2
/home/welch/install/man
```

If you want to have installations for several different platforms, then specify an `--exec-prefix` that is different for each platform. For example, if you use `--exec-prefix=/home/welch/install/solaris`, then the *tclsh8.2* program and `libtcl8.2.so` shared library will be installed in:

```
/home/welch/install/solaris/bin/tclsh8.2
/home/welch/install/solaris/lib/libtclsh8.2.so
```

The script libraries and manual pages will remain where they are, so they are shared by all platforms. Note that Windows has a slightly different installation location for binary libraries. They go into the `*arch*/bin` directory along with the main executable programs.

Using Stub Libraries

One problem with extensions is that they get compiled for a particular version of Tcl. As new Tcl releases occur, you find yourself having to recompile extensions. This was necessary for two reasons. First, the Tcl C library tended to change its APIs from release to release. Changes in its symbol table tie a compiled extension to a specific version of the Tcl library. Another problem occurred if you compiled *tclsh* statically and then tried to dynamically load a library. Some systems do not support back linking in this situation, so *tclsh* would crash. Paul Duffin created a *stub library* mechanism for Tcl that helps solve these problems.

The main idea is that Tcl creates two binary libraries: the main library (e.g., `libtcl8.2.so`) and a stub library (e.g., `libtclstub.a`). All the code is in the main library. The stub library is just a big jump table that contains addresses of functions in the main library. An extension calls Tcl through the jump table. The level of indirection makes the extension immune to changes in the Tcl library. It also handles the back linking problem. If this sounds expensive, it turns out to be equivalent to what the operating system does when you use shared libraries (i.e., dynamic link libraries). Tcl has just implemented dynamic linking in a portable, robust way.

To make your extension use stubs, you have to compile with the correct flags, and you must add a new call to your extensions `Init` procedure (e.g., Examplea_Init). The TCL_USE_STUBS compile-time flag turns the Tcl C API calls into macros that use the stub table. The Tcl_InitStubs call ensures that the jump table is initialized, so you must call Tcl_InitStubs as the very first thing in your Init procedure. A typical call looks like this:

```
if (Tcl_InitStubs(interp, "8.1", 0) == NULL) {
    return TCL_ERROR;
}
```

Tcl_InitStubs is similar in spirit to Tcl_PkgRequire in that you request a minimum Tcl version number. Stubs have been supported since Tcl 8.1, and the API will evolve in a backward-compatible way. Unless your extension uses new C APIs introduced in later versions, you should specify the lowest version possible so that it is compatible with more Tcl applications.

Using autoconf

Autoconf uses the *m4* macro processor to translate the configure.in template into the *configure* script. The *configure* script is run by /bin/sh (i.e., the Bourne Shell). Creating the configure.in template is simplified by a standard *m4* macro library that is distributed with *autoconf*. In addition, a Tcl distribution contains a tcl.m4 file that has additional *autoconf* macros. Among other things, these macros support the standard *configure* flags described in Table 45–3.

Creating configure templates can be complex and confusing. Mostly this is because a macro processor is being used to create a shell script. The days I have spent trying to change the Tcl configuration files really made me appreciate the simplicity of Tcl! Fortunately, there is now a standard set of Tcl-specific *autoconf* macros and a sample Tcl extension that uses them. By editing the configure.in and Makefile.in sample templates, you can ignore the details of what is happening under the covers.

The tcl.m4 File

The Tcl source distribution includes tcl.m4 and aclocal.m4 files. The *autoconf* program looks for the aclocal.m4 file in the same directory as the configure.in template. In our case, the aclocal.m4 file just includes the tcl.m4 file. *Autoconf* also has a standard library of m4 macros as part of its distribution. To use tcl.m4 for your own extension, you have some options. The most common way is to copy tcl.m4 into aclocal.m4 in your source directory. Or you can copy both aclocal.m4 and tcl.m4 and have aclocal.m4 include tcl.m4. If necessary, you can add more custom macros to this aclocal.m4 file. Alternatively, you can copy the tcl.m4 file into the standard *autoconf* macro library.

The tcl.m4 file defines macros whose names begin with SC_ (for Scriptics). The standard *autoconf* macro names begin with AC_. This book does not provide an exhaustive explanation of all these *autoconf* macros. Instead, the important ones are explained in the context of the sample extension.

The tcl.m4 file replaces the tclConfig.sh found in previous versions of Tcl. (Actually, tclConfig.sh is still produced by the Tcl 8.2 *configure* script, but its use is deprecated.) The idea of tclConfig.sh was to capture some important results of Tcl's *configure* so that they could be included in the *configure* scripts used by an extension. However, it is better to recompute these settings when configuring an extension because, for example, different compilers could be used to build Tcl and the extension. So, instead of including tclConfig.sh into an extension's *configure* script, the extension's configure.in should use the SC_ macros defined in the *tcl.m4* file.

Makefile Templates

Autoconf implements yet another macro mechanism for the Makefile.in templates. The basic idea is that the *configure* script sets shell variables as it learns things about your system. Finally, it substitutes these variables into Makefile.in to create the working Makefile. The syntax for the substitutions in Makefile.in is:

```
@configure_variable_name@
```

Often, the *make* variable and the shell variable have the same name. For example, the following statement in Makefile.in passes the TCL_LIBRARY value determined by *configure* through to the Makefile:

```
TCL_LIBRARY = @TCL_LIBRARY@
```

The AC_SUBST macro specifies what shell variables should be substituted in the Makefile.in template. For example:

```
AC_SUBST(TCL_LIBRARY)
```

The Sample Extension

This section describes the sample extension that is distributed as part of the Tcl Extension Architecture (TEA) standard. The goal of TEA is to create a standard for Tcl extensions that makes it easier to build, install, and share Tcl extensions. The sample Tcl extension is on the CD, and it can be found on the Web at:

```
ftp://ftp.scriptics.com/pub/tcl/examples/tea/
```

There is also documentation on the Web at:

```
http://www.scriptics.com/products/tcltk/tea/
```

The extension described here is stored in the network CVS repository under the module name samplextension. If you want direct access to the latest versions of Tcl source code, you can learn about the CVS repository at this web page:

```
http://www.scriptics.com/products/tcltk/netcvs.html
```

The sample extension implements the Secure Hash Algorithm (SHA1). Steve Reid wrote the original SHA1 C code, and Dave Dykstra wrote the original Tcl interface to it. Michael Thomas created the standard configure and Makefile templates.

Instead of using the original name, `sha1`, the example uses a more generic name, `exampleA`, in its files, libraries, and package names. When editing the sample templates for your own extension, you can simply replace occurrences of "exampleA" with the appropriate name for your extension. The sample files are well commented, so it is easy to see where you need to make the changes.

configure.in

The `configure.in` file is the template for the *configure* script. This file is very well commented. The places you need to change are marked with `__CHANGE__`. The first macro to change is:

```
AC_INIT(exampleA.h)
```

The `AC_INIT` macro lists a file that is part of the distribution. The name is relative to the `configure.in` file. Other possibilities include `../generic/tcl.h` or `src/mylib.h`, depending on where the configure.in file is relative to your sources. The `AC_INIT` macro necessary to support building the package in different directories (e.g., either `tcl8.2/unix` or `tcl8.2/unix/solaris`). The next thing in `configure.in` is a set of variable assignments that define the package's name and version number:

```
PACKAGE = exampleA
MAJOR_VERSION = 0
MINOR_VERSION = 2
PATCH_LEVEL =
```

The package name determines the file names used for the directory and the binary library file created by the Makefile. This name is also used in several configure and Makefile variables. You will need to change all references to "exampleA" to match the name you choose for your package.

The version and patch level support a three-level scheme, but you can leave the patch level empty for two-level versions like `0.2`. If you do specify a patchlevel, you need to include a leading "." or "p" in it. These values are combined to create the version number like this:

```
VERSION = ${MAJOR_VERSION}.${MINOR_VERSION}${PATCH_LEVEL}
```

Windows compilers create a special case for shared libraries (i.e., DLLs). When you compile the library itself, you need to declare its functions one way. When you compile code that uses the library, you need to declare its functions another way. This complicates the `exampleA.h` header file. Happily, the complexity is hidden inside some macros. In `configure.in`, you simply define a `build_`*`Package`* variable. The sample defines:

```
AC_DEFINE(BUILD_exampleA)
```

This variable is set only when you are building the library itself, and it is defined only when compiling on Windows. We will show later how this is used in `exampleA.h` to control the definition of the `Examplea_Init` procedure.

The `configure.in` file has a bunch of magic to determine the name of the shared library file (e.g., `packageA02.dll`, `packageA.0.2.so`, `packa-`

`geA.0.2.shlib`, etc.). All you need to do is change one macro to match your package name.

```
AC_SUBST(exampleA_LIB_FILE)
```

These should be the only places you need to edit when adapting the sample `configure.in` to your extension. It is worth noting that the last macro determines which templates are processed by the configure script. The sample generates two files from templates, `Makefile` and `mkIndex.tcl`:

```
AC_OUTPUT([Makefile mkIndex.tcl])
```

The `mkIndex.tcl` script is a script that runs `pkg_mkIndex` to generate the `pkgIndex.tcl` file. The `pkg_mkIndex` command is described in Chapter 12. The `mkIndex.tcl` script is more complex than you might expect because UNIX systems and Windows have different default locations for binary libraries. The goal is to create a `pkgIndex.tcl` script that gets installed into the *arch*`/lib/`*package* directory but that can find either *arch*`/bin/`*package*`.dll` or *arch*`/lib/lib`*package*`.so`, depending on the system. You may need to edit the `mkIndex.tcl.in` template, especially if your extension is made of both Tcl scripts and a binary library.

Makefile.in

The `Makefile.in` template is converted by the *configure* script into the `Makefile`. The sample `Makefile.in` is well commented so that it is easy to see where to make changes. However, there is some first class trickery done with the Makefile variables that is not worth explaining in detail. (Not in a Tcl book, at least!) There are a few variables with `exampleA` in their name. In particular, `exampleA_LIB_FILE` corresponds to a variable name in the configure script. You need to change both files consistently. Some of the lines you need to change are shown below:

```
exampleA_LIB_FILE = @exampleA_LIB_FILE@
lib_BINARIES = $(exampleA_LIB_FILE)
$(exampleA_LIB_FILE)_OBJECTS = $(exampleA_OBJECTS)
```

The `@`*varname*`@` syntax is used to substitute the configure variable with its platform-specific name (e.g., `libexamplea.dll` or `libexample.so`). The `lib_BINARIES` variable names the set of libraries built by the "make binaries" target. The `_OBJECT` variable is a clever trick to allow a generic library make rule, which appears in the `Makefile.in` template as `@MAKE_LIB@`. Towards the end of `Makefile.in`, there is a rule that uses these variables, and you must change uses of `exampleA` to match your package name:

```
$(exampleA_LIB_FILE) : $(exampleA_OBJECTS)
        -rm -f $(exampleA_LIB_FILE)
        @MAKE_LIB@
        $(RANLIB) $(exampleA_LIB_FILE)
```

You must define the set of source files and the corresponding object files that are part of the library. In the sample, `exampleA.c` implements the core of the Secure Hash Algorithm, and the `tclexampleA.c` file implements the Tcl command interface:

```
exampleA_SOURCES = exampleA.c tclexampleA.c
SOURCES = $(exampleA_SOURCES)
```

The object file definitions use the `OBJEXT` variable that is `.o` for UNIX and `.obj` for Windows:

```
exampleA_OBJECTS = exampleA.${OBJEXT} tclexampleA.${OBJEXT}
OBJECTS = $(exampleA_OBJECTS)
```

The header files that you want to have installed are assigned to the `GENERIC_HDRS` variable. The `srcdir` Make variable is defined during *configure* to be the name of the directory containing the file named in the `AC_INIT` macro:

```
GENERIC_HDRS = $(srcdir)/exampleA.h
```

Unfortunately, you must specify explicit rules for each C source file. The `VPATH` mechanism is not reliable enough to find the correct source files reliably. The *configure* script uses `AC_INIT` to locate source files, and you create rules that use the resulting `$(srcdir)` value. The rules look like this:

```
exampleA.$(OBJEXT) : $(srcdir)/exampleA.c
	$(COMPILE) -c ‘@CYGPATH@ $(srcdir)/exampleA.c‘ -o $@
```

The sample Makefile includes several standard targets. Even if you decide not to use the sample `Makefile.in` template, you should still define the targets listed in Table 45–4 to ensure your extension is TEA compliant. Plans for automatic build environments depend on every extension implementing the standard make targets. The targets can be empty, but you should define them so that *make* will not complain if they are used.

Table 45–4 TEA standard Makefile targets.

`all`	Makes these targets in order: `binaries`, `libraries`, `doc`.
`binaries`	Makes executable programs and binary libraries (e.g., DLLs).
`libraries`	Makes platform-independent libraries.
`doc`	Generates documentation files.
`install`	Makes these targets in order: `install-binaries`, `install-libraries`, `install-doc`.
`install-binaries`	Installs programs and binary libraries.
`install-libraries`	Installs script libraries.
`install-doc`	Installs documentation files.
`test`	Runs the test suite for the package.
`depend`	Generates makefile dependency rules.
`clean`	Removes files built during the make process.
`distclean`	Removes files built during the configure process.

Standard Header Files

This section explains a technique you should use to get symbols defined properly in your binary library. The issue is raised by Windows compilers, which have a notion of explicitly importing and exporting symbols. When you build a library you export symbols. When you link against a library, you import symbols. The BUILD_exampleA variable is defined on Windows when you are building the library. This variable should be undefined on UNIX, which does not have this issue. Your header file uses this variable like this:

```
#ifdef BUILD_exampleA
#undef TCL_STORAGE_CLASS
#define TCL_STORAGE_CLASS DLLEXPORT
#endif
```

The TCL_STORAGE_CLASS variable is used in the definition of the EXTERN macro. You must use EXTERN before the prototype for any function you want to export from your library:

```
EXTERN int Examplea_Init _ANSI_ARGS_((Tcl_Interp *Interp));
```

The _ANSI_ARGS_ macro is used to guard against old C compilers that do not tolerate function prototypes.

Using the Sample Extension

You should be able to configure, compile, and install the sample extension without modification. On my Solaris machine, the binary library is named exampleA0.2.so, while on my Windows NT machine the library is named exampleA02.dll. The package name is Tclsha1, and it implements the sha1 Tcl command. Ordinarily these names would be more consistent with the file names and package names in the template files. However, the names in the sample are designed to be easy to edit in the template. Assuming that you use make install to copy the binary library into the standard location for your site, you can use the package from Tcl like this:

```
package require Tclsha1
sha1 -string "some string"
```

The sha1 command returns a 128 bit encoded hash function of the input string. There are a number of options to sha1 you can learn about by reading the manual page that is included with the extension.

Blank page 652