The Tcl Extension Architecture

Brent Welch <welch@scriptics.com>
Michael Thomas <wart@scriptics.com>
Scriptics Corporation

Abstract

This paper describes goals and current state of the Tcl Extension Architecture (TEA). The goal of TEA is to create a standard for Tcl extensions that makes it easier to build, install, and share Tcl extensions. In its current form, TEA specifies a standard compilation environment for Tcl and its extensions. The standard uses `autoconf`, `configure` and `make` on UNIX and Windows. A longer term goal is to create an infrastructure that supports network distribution and installation of Tcl extensions. A standard build environment is a necessary first step to support automated compilation and distribution of extensions. This paper describes the current state of TEA, but we expect to continue to refine the standard and add to it as we gain experience with it.

Introduction

Compiling Tcl from the source distribution is easy. One of the strengths of Tcl is that it is quite portable and 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. TEA covers the following topics, which are described in more detail in the paper:

- Recommended Source Directory Structure:
- Standard Installation Directory Structure.
- Stubs Libraries.
- `Autoconf` and `Configure`.
- Standard `Make` Targets.
- A Sample TEA-Compliant Extension.
- Future Plans.

Standard Directory Structure

One goal of TEA is to make the process of configuring and building a Tcl extension very similar to building Tcl itself. In addition, building a Tcl extension depends on having access to the Tcl source distribution. You must configure and build Tcl before you build your extensions. The best way to organize your source code is to have Tcl and all your extensions under a common directory (e.g., `/usr/local/src` or `/home/welch/cvs`). This way the build process for an extension can automatically find the Tcl sources. The dependency on the Tcl source distribution is described later, and in the long term we hope to support building TEA-compliant extensions against a binary distribution of Tcl.

The Source Distribution

Table 1 describes the directory structure of the Tcl source distribution. The Tk distribution is similar. The directory structure divides the sources into generic and platform-specific directories.

Table 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 *`platform`* 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*.
`tcl8.2/win/`*`Build`*	*`Build`* is `Release` or `Debug`. This contains compiler output.

The Installation Directory Structure

When you install Tcl, the files end up in a different arrangement than the one in the source distribution. The standard installation directory is organized so 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`. The --prefix option specifies the root of the installation directory (e.g., /usr/local). The --exec-prefix option specifies a platform-specific directory (e.g., /usr/local/solaris-sparc) for applications and programming libraries. Table 2 shows the standard installation directory structure:

Table 2 The installation directory structure relative to the `--prefix` directory.

exec_prefix`/bin`	This contains platform-specific applications. On Windows, this also contains binary libraries (i.e., DLLs). Typical *exec_prefix* names end with `solaris-sparc`, `linux-ix86`, and `win-ix86`.
exec_prefix`/lib`	This contains platform-specific binary libraries on UNIX systems (e.g., `libtcl8.2.so`)
exec_prefix`/lib/`*package*	Contains `pkgIndex.tcl` files corresponding to binary libraries from *package* that are found in *exec_prefix*`/lib`.
`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.
`lib/tcl8.2`	This contains the contents of the `tcl8.2/library` source directory, including subdirectories.
`lib/`*package*	This contains Tcl scripts for *package* and its `pkgIndex.tcl` file. Example *package* directories include `tk8.2` and `itcl3.0.1`.
`man`	This contains reference documentation in UNIX *man* format.

The Package Mechanism

Extensions are installed and used as packages. A package can be one Tcl script, a collection of Tcl scripts, a binary library, or some combination of scripts and libraries. When you install an extension you need to update the package registry so that others can find the extension with `package require`. This section describes the default package management system, which uses a collection of pkgIndex.tcl files in directories along your auto_path.

The package registry is implemented by a collection of pkgIndex.tcl files. Tcl searches the directories listed in its auto_path variable for pkgIndex.tcl files. It also searches down one directory, so you can put your extensions and pkgIndex.tcl files into subdirectories of the main directories listed on auto_path. The default auto_path is

prefix/lib/tcl*version prefix*/lib *exec_prefix*/lib

Each pkgIndex.tcl file has one or more `package ifneeded` commands in it. These register Tcl commands that are called whenever a particular package is requested with `package require`. This section shows a few sample `package ifneeded` scripts to handle different configurations of packages.

Binary Library

A binary library (i.e., DLL) goes into the platform-specific lib directory. For example, you install your DLL into *exec_prefix*/lib/libfoobar1.0.so and you create a package index file in *exec_prefix*/lib/foobar1.0/pkgIndex.tcl, which contains this code:

```
package ifneeded foobar 1.0 [list load \
    [file join $dir .. \
libfoobar1.0[info sharedlibextension]]\
    Foobar]
```

Collecting all the binary libraries in one directory makes it easy to resolve dependencies among them and third-party libraries that support your Tcl extension. UNIX users may have to adjust their `LD_LIBRARY_PATH` to include the *exec_prefix*/lib directory. On Windows, these files are actually in the *exec_prefix*/bin directory, which is automatically searched. Keeping the pkgIndex.tcl files in separate directories keeps them independent.

Tcl Scripts

If your extension is just Tcl scripts, then it can be shared by users on different platforms. These libraries are typically kept in a subdirectory of *prefix*/lib, (e.g., /usr/local/lib/foobar1.0). You can use the `pkg_mkIndex` Tcl command to generate a pkgIndex.tcl file for your scripts:

```
pkg_mkIndex -verbose prefix/lib *.tcl
```

By default, pkg_mkIndex generates pkgIndex.tcl files that contain `tclPkgSetup` commands that use `source` or `load` indirectly. You might imagine that `package require` actually loads code, but by default is does not. Instead, the following `tclPkgSetup` command arranges for foobar.tcl to be sourced whenever the unknown command tries to find `Foobar_Init`, `Foobar_DoSomething`, or `Foobar_End`.

```
package ifneeded foobar 1.0 \
    [list tclPkgSetup $dir foobar 1.0 \
   {{foobar.tcl source {Foobar_Init
    Foobar_DoSomthing Foobar_End}}}]
```

The tclPkgSetup command is complex, so you should use the `pkg_mkIndex` command to generate these commands for you. If you use `pkg_mkindex -direct`, you can create a simpler package that is sourced immediately in response to the `package require` command. This direct package index looks like this:

```
package ifneeded foobar 1.0 \
    [list source [file join $dir
foobar.tcl]]
```

Library and Script Combination

If you have both scripts and binary libraries, then you can split your package into two parts: the shared part as Tcl scripts, and a platform-specific part as a binary library. The tricky part is building your pkgIndex.tcl file correctly. There are two problems. First, you can only have one `package ifneeded` command for a single package, so you need to specify something about the scripts and the library in one command. Next, you cannot predict the location of both parts of the package, so you have to assume they are installed in a standard location relative to the auto_path. Our preferred solution is modeled after the one in the SNACK sound extension by Kåre Sjölander.

Create a pkgIndex.tcl file in the *exec_prefix*/lib/foobar1.0 subdirectory. You will need to install a copy for each different platform that you compile for (e.g., solaris-sparc/lib/foobar1.0/pkgIndex.tcl and linux-ix86/lib/foobar1.0/pkgIndex.tcl.). This file loads the binary library and sources the Tcl script. We assume that the scripts are installed relative to the Tcl script library:

```
package ifneeded foobarArch 1.0 \
    "[list load \
    [file join $dir ../libfoobar1.0[info \
      sharedlibextension] Foobar] \;
    [list source [file join [file dirname \
      $tcl_library] \
        foobar1.0/foobar.tcl]]"
```

This example assumes the standard directory structure. It would be more general to search along the auto_path for the foobar1.0 subdirectory and then source its foobar.tcl file. If you have several script files, you can introduce a short procedure to source all of them. Or, you can have two pkgIndex.tcl files and require that your users require both (e.g., foobar and foobarArch). The packages can also require each other. For example:

```
package ifneeded foobarArch 1.0 \
    "[list load \
    [file join $dir ../libfoobar1.0[info \
      sharedlibextension] Foobar] \;
    [list package require foobar 1.0]"
```

Finally, by using the package unknown hook, you could define and use an alternate package manager. Newsgroup discussions have pointed out that searching for all the pkgIndex.tcl files can be slow on some systems. An alternate package manager could keep a more compact and efficient database, and perhaps have smarts about downloading packages from standard TEA repositories.

Autoconf, Configure and Make

In the past, UNIX, Windows, and Macintosh have different compilation environments. The advent of the free Cygwin tools have made it possible to standardize on `autoconf`, `configure` and `make` for the UNIX and Windows compilation environments. The Macintosh still uses Code Warrior project files, however. On Windows we use `make`, `sh`, and `autoconf` from Cygwin, and the `cl` (VC++) compiler from Microsoft.

The autoconf system is used to create Makefiles that have settings appropriate for the current operating system. 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 configure script, in turn, is used to generate the working Makefile. The three steps: setup, configuration and make, are illustrated by the build process for Tcl and Tk:

1. 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.
2. A user of a source code package runs configure on the computer system they will use to compile the sources. The configure script examines the current system and makes various settings that are used during compilation.

Table 3 Standard configure flags.

--prefix=*dir*	This 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	Use the *gcc* compiler instead of the default system compiler.
--disable-shared	Disable generation of shared libraries and Tcl shells that dynamically link against them. Statically linked shells and static archives are built instead.
--enable-symbols	Compile with debugging symbols.
--enable-threads	Compile with thread support turned on.
--with-tcl=*dir*	This specifies the location of the build directory for Tcl.
--with-tk=*dir*	This specifies the location of the build directory for Tk.
--with-tclinclude=*dir*	This specifics the directory that contains tcl.h.
--with-tcllib=dir	This specifies the directory that contains the Tcl binary library (e.g., libtclstubs.a). (*Note*: this option is not yet supported.)
--with-x-includes=dir	This specifics the directory that contains X11.h.
--with-x-libraries=dir	This specifies the directory that contains the X11 binary library (e.g., libX11.6.0.so).

3. 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. This step converts Makefile.in to a Makefile suitable for the platform and configuration settings. .

4. Once `configure` is complete, you build your program with `make`. This steps checks your source files against the compiled files and reruns the compiler on any files that have changed since the last compilation. The results are binary libraries for extensions and executable programs for applications. `Make` is used for testing and installation, too. Table 5 on page 8 shows the standard `make` targets.

Standard `configure` Flags

Table 3 shows the standard options for Tcl `configure` scripts. These are implemented by a `configure` library file (`tcl.m4`) that you can use in your own `configure` scripts. The facilities provided by `tcl.m4` are described in more detail later. There are also many other command line options that come standard with `configure`. Some of these are meant to give you control over where the different parts of the installation go. However, because of the way Tcl automatically searches for scripts and binary libraries, you can mess up the Tcl installation by installing the libraries and the binaries in wildly different locations. Because of this, the Tcl installation procedures in the standard Makefile do not support the `--libdir` and `--bindir` options. In general, if the flags are not listed in Table 3, then they are not guaranteed to be supported by the standard Makefile template.

Examples

If you only have one platform, simply run `configure` in the `unix` (or `win`) directory:

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

Use `./configure` to ensure you run the `configure` script from the current directory. If you build for multiple platforms, create subdirectories of `unix` and run `configure` from there. You are free to create the compilation directory anywhere (some prefer to keep all the generated files away from the sources.) Here we just use a subdirectory of the unix directory:

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

Any flag with `disable` or `enable` in its name can be inverted. Table 3 lists the non-default 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-pre-
fix=/home/welch/install/solaris \
    --enable-gcc --enable-threads --
enable-symbols
```

Your builds will go the most smoothly if you organize all your sources under a common directory. In this case, you should be able to 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 everything gets installed together.

If you use alternate build directories, like the unix/solaris example above, you must specify `--with-tcl` when building your extensions. This is the directory where the Tcl build occurred. It contains libraries and a tclConfig.sh file that is used by the extensions configure process.

If your source tree is not adjacent to the Tcl source tree, then you must use `--with-tclinclude` or `--with-tcllib` so 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-x-includes` and `--with-x-libraries` flags are similar options necessary when building Tk if your X11 installation is in a non-standard location.

Finding 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 `configure` has failed to find a working compiler. In the first case, it assumes 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. The cross-compiling message typically occurs when your UNIX environment isn't set up right to find the compiler.

On Windows there is a more explicit compiler check, and `configure` exits if it cannot find the compiler. Currently, the Windows configure macros knows only about the Visual C++ compiler. VC++ ships with a batch file, `vcvars32.bat`, that sets up the environment so you can run the compiler, `cl`, from the command line. You must run `vcvars32.bat` before running `configure`, or set up your environment so you do not have to remember to run the batch file all the time.

Installation Directories

The `--prefix` flag specifies the main installation directory (e.g., `/home/welch/install`). The directories listed in Table 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 (i.e., DLLs). They go into the *`exec_prefix`*`/bin` directory along with the main executable programs.:

Table 4 Standard autoconf macros defined by `tcl.m4`.

`SC_PATH_TCLCONFIG`	Locate the `tclConfig.sh` file and sanity check the compiler flags. This implements the `--with-tcl` option.
`SC_PATH_TKCONFIG`	Locate the `tkConfig.sh` file. This implements `--with-tk`.
`SC_LOAD_TCLCONFIG`	Load the tclConfig.sh file.
`SC_LOAD_TKCONFIG`	Load the tkConfig.sh file.
`SC_ENABLE_GCC`	Implements the --enable-gcc option.
`SC_ENABLE_SHARED`	Implements the --enable-shared option.
`SC_ENABLE_THREADS`	Implements the --enable-threads option.
`SC_ENABLE_SYMBOLS`	Implements the --enable-symbols option.
`SC_MAKE_LIB`	Generates definitions used to make shared or unshared libraries on various platforms.
`SC_LIB_SPEC`	Generates the name of a library and appropriate linker flags needed to link it.
`SC_PRIVATE_TCL_HEADERS`	Use this if you need to include `tclInt.h`
`SC_PUBLIC_TCL_HEADERS`	Locate the standard Tcl include files.

Using autoconf and the tcl.m4 File

Autoconf uses the m4 macro processor to translate the configure.in template into the configure script. 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 3.

The goal of tcl.m4 is to simply the configure.in templates used for extensions and to replace the use of the tclConfig.sh and tkConfig.sh files. The idea of tclConfig.sh was to capture some important results of Tcl's configure so 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. At present Tcl still generates tclConfig.sh, and some of the tcl.m4 macros depend on it. We plan to restructure the macros further so tclConfig.sh (and tkConfig.sh) will no longer be needed. So, instead of using SC_LOAD_TCLCONFIG, extensions will use a new macro that computes compiler settings.

Table 4 lists the public macros defined in the tcl.m4 file. The tcl.m4 file defines macros whose names begin with SC_ (for Scriptics). The four TCLCONFIG and TK_CONFIG macros listed in Table 4 will be eventually be replaced. There are other macros defined, but the following are the only ones guaranteed to persistStandard Make Targets
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 5 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 5 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: all, install-binaries, install-libraries, install-doc.
install-binaries	Makes binaries, and installs programs and binary libraries.
install-libraries	Makes libraries, and installs script libraries.
install-doc	Makes doc, and 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	Makes clean, and removes files built during the configure process.

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 changes 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 contains a big jump table that has addresses of the 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 have to 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 Tcl C 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.

The Sample Extension

This section describes the sample extension that is distributed as part of TEA. 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.

The goal of the sample extension is to provide a TEA-compliant example that is easy to read and modify for your own extension. 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 is 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 patch-level, 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. The standard configure.in defines a `build_`*`Package`* variable with the following line, which you do not need to change:

```
AC_DEFINE_UNQUOTED(BUILD_${PACKAGE})
```

The `build_packageA` variable is only set when you are building the library itself, and it is only defined 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`, `packageA.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.

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. 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)
```

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 tclexam-
pleA.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 `cygpath` program converts file names to different formats required by different tools on Windows. On UNIX, the `CYGWIN` macro is simply defined to `echo`.

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 it creates a binary library 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 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 man page that is part of the sample.

Future Directions

The short term goal of TEA is to provide a standard way to build Tcl extensions. We have created a sample extension for others to learn from, and we have been converting the widely used [incr Tcl], Expect, and TclX extensions to adhere to the standard.

The long term goal of TEA is to make distributing and installing Tcl extensions easy for the end user. We envision a system where open source extensions can be hosted in a common CVS repository, built automatically on a variety of platforms, and distributed to end users and installed automatically on their system. For this goal to succeed, we need to start with a standard framework for configuring and building extensions.

Web Links

The TEA home page is:

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

The sample extension can be found at the Scriptics FTP site:

```
ftp://ftp.scriptics.com/pub/tcl/exam-
ples/tea/
```

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

```
http://www.scriptics.com/cvs/
```

Acknowledgments

We would like to thank the following people and organizations: Paul Duffin, Jan Nijtmans, Jean-Claude Wippler, and Scott Stanton designed and implemented the stub library mechanism for Tcl. Steve Reid and Dave Dykstra wrote the Secure Hash Algorithm and the Tcl interface to it. Unified building procedures on all flavors of UNIX would not be possible without the autoconf tools from the GNU project, and the Cygwin tools from Cygwin extend this functionality to Windows.

A different version of this paper will appear as a chapter in Brent Welch's book, *Practical Programming in Tcl and Tk*, 3rd Edition, published by Prentice Hall.