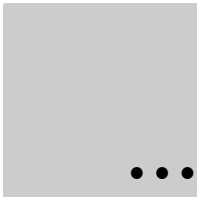

Tcl Extension Architecture Developer's Guide — DRAFT

Scriptics Corporation
August 25, 1999

Scriptics Corporation
2593 Coast Avenue
Second Floor
Mountain View, CA 94043
U.S.A

http://www.scriptics.com

Contents

Preface

Tcl (Tool Command Language, often pronounced “tickle”) is a freely-available scripting language that’s designed to be extensible. Extending Tcl means more than writing new Tcl procedures using Tcl itself. You can add new functionality, creating new Tcl commands, by writing code in C or C++. (A package called TclBlend lets you extend Tcl by writing Java.) Thousands of extensions have already been written. A few of the most popular extensions include Tk (for graphical interfaces), Expect (for automating interaction with other programs), and [incr Tcl] (object-oriented Tcl).

Although Tcl is extensible, there’s never been a standard recommended way to write an extension. So, for instance, different extensions have different build procedures. Libraries are installed in assorted places, and are built in various ways as extensions are ported to various platforms. Some APIs haven’t been created or exported for extension writers—so, for example, some Tcl extensions have used internal APIs that required modifications to the Tcl core itself. All of this means extra work and headaches for system administrators, users, and programmers.

TEA, the Tcl Extension Architecture, is a proposed standard for writing, building and documenting Tcl extensions. TEA is far from complete. It needs input from extension writers and users to become a standard that helps everyone.

This document also is incomplete. Scriptics Corporation (which distributes Tcl) will work with the community to revise and complete this document and the TEA standard. Please help us by reading the latest version of this document; there’s a link from *http://www.scriptics.com/products/tcltk/tea/*. You can discuss TEA, and this document, on the mailing list *tea@scriptics.com*. To subscribe to the list, send a message to *tea-request@scriptics.com* with the single word *subscribe* in the message body.

Chapter 1
Overview

TEA, the Tcl Extension Architecture, is a draft standard that's under development. As of August 1999, TEA covers only the required configuration and Makefile settings. The proposed goal for TEA is a universal architecture for Tcl extensions. An end user should be able to go to a Tcl extension server (similar to the Perl CPAN archives) and use a GUI to choose extensions they want. The extensions will conform to the TEA API, which means that the user should be able either to download a binary file and install it automatically or download sources and install them almost as easily.

For more about TEA, see *http://www.scriptics.com/products/tcltk/tea/*. You can download the sample extension from *ftp://ftp.scriptics.com/pub/tcl/examples/tea/*.

Like TEA itself, **this document is a rough draft**. Some chapters are basically skeletons, placeholders for information that may be added in the future.

For an overview of how to write an extension and how it integrates with Tcl, see **Chapter 2**. It has more information, including the recommended filesystem locations for an extension.

Chapter 3 has a brief description of recommended coding styles for Tcl and C.

Collections of Tcl commands are kept in *libraries* and organized into *packages*. Packages support version numbers and have a *provide/require* model of use. Read more in **Chapter 4**.

Tcl Stubs solve several problems with extensions, including conflicts between static vs. dynamic loading and recompilation when new versions of Tcl are released. See **Chapter 5** for details.

TEA standardizes configuration and build tools. You'll need to modify two template files that control the build process: *configure.in* and *Makefile.in*. **Chapter 6** describes these files and the changes needed.

Chapter 7 explains how to create a test suite for your extension.

Chapter 8 discusses writing documentation, naming and storing the files.

Appendix A describes each required Configure switch defined in the *tcl.m4* file.

Chapter 2
Designing and Coding Tcl Extensions

[This document is a rough draft. Here we're planning to add summary info on the basics of writing extensions—enough to get the context without the nitty-gritty details.]

Directory Structure

TEA recommends directory structures for extensions but doesn't require any one of them.

[We're thinking of describing two suggested hierarchies here: 1) Tcl-only, 2) Platform-independent (Tcl plus C). Two other possible hierarchies are: 3) Platform-dependent (UNIX and Windows), and especially 4) How to build static shells (Tcl & Tk).]

Chapter 3
Recommended Coding Style

We have recommended (not required) coding guidelines for Tcl. These can make your code easier to understand and more maintainable. See *http://www.scriptics.com/doc/styleGuide.pdf*.

Another good source of coding information for C programming is the Tcl Engineering Manual at *http://www.scriptics.com/doc/engManual.ps*.

[This document is a rough draft. We'll add highlights from those documents here. We also should update those documents and/or put updates here.]

Chapter 4
Tcl Packages

[This document is a rough draft. Here, the document could describe packages and the benefits of using them. It can also explain the namespace facility. The Style Guide has recommendations for coding packages; see *http://www.scriptics.com/doc/styleGuide.pdf*.]

Package names: The name must not end with a digit (0 through 9). [?Rules for other special characters like () {} ‘’ “”?] This name should be unique; no other extension should have the same one. The NIST registry of Tcl packages at *http://pitch.nist.gov/nics/* lists some Tcl extensions. You also can search the Tcl Resource Center on the Scriptics website, *http://www.scriptics.com/resource/*, to see if anyone else has used the name you want.

[Here we’re planning to explain package rules. For example:

- Naming (initial caps vs. all lowercase)
- Namespace and name of package
- Define declarations in namespace eval
- Define all procs in global space
- How to import/export

...]

Chapter 5
Tcl Stubs

[This document is a rough draft. The plans for this chapter include:

The benefits of using stubs, an overview of how they work.

1) Extension writers should link against the Tcl stub library, not against Tcl itself.

   For more information, see the document *How To Use Stub Libraries* at *http://www.scriptics.com/services/support/howto/stubs.html*.

2) How to stub-enable extensions.]

Chapter 6
Configure and Make Files

The extension writer must supply customized configuration files named *configure.in* and *Makefile.in*, modified from templates. These two template files, plus a stock library of Autoconf macros in the file *tcl.m4*, are supplied with the Tcl Sample Extension. This section describes those three files. You can get a copy from the directory *ftp://ftp.scriptics.com/pub/tcl/examples/tea/*.

Quick Overview

This section is a high-level summary for extension writers who are familiar with Autoconf, Cygwin and Makefiles. You may be able to "TEA-ify" your extension simply by reading this section. If you need more details, though, scan through the rest of this Chapter.

The *tcl.m4* file defines standard TEA Configure switches. To add more switches, you can either edit the *configure.in* file or add another M4 file. You must append *tcl.m4* (and any other M4 file you create) to the *aclocal.m4* file before running **autoconf**.

The *configure.in* and *Makefile.in* files have well-marked places at the start, middle and end of each file that you'll need to edit for your extension. The main kinds of changes are:

- Replace the sample extension name "exampleA," wherever it occurs, in upper or lower case, with the name of your extension.
- Update the macros and targets that list the files from the sample extension (*exampleA.** and *tclexampleA.**) to list the files from your extension.

Finally, please help us make TEA into an architecture that works for everyone by discussing your experiences on the TEA mailing list. If you define new M4 macros, send them in! This document's Preface has details.

Programs You'll Use

To write a TEA-compliant extension, a programmer uses the GNU Autoconf package; this generates a **configure** script for the end user. To install an extension, the end user runs the **configure** script, then runs a compatible version of **make**.

- These freely available utilities, from *http://www.gnu.org/*, are common on UNIX and UNIX-like systems.
- On Microsoft Windows platforms, these utilities are part of the freely available Cygwin package. To install and use Cygwin, follow these steps:
  a) Download and install the Cygnus Cygwin user tools package from *ftp://go.cygnus.com/pub/sourceware.cygnus.com/pub/cygwin/cygwin-b20/usertools.exe.*
  b) Install GNU Make for Windows. You can get the GNU **make** binary from the same directory as the Tcl Sample Extension: *ftp://ftp.scriptics.com/pub/tcl/example/*
  c) Create a directory called *C:\bin*. Copy the *sh.exe* program from the Cygnus *bin* directory (*C:\cygnus\cygwin-b20\H-i586-cygwin32\bin\sh.exe*) to *C:\bin*. This will allow you to run shell scripts that use the "#!/bin/sh" invocation.
  d) Run *vcvars32.bat*. You must run that batch file every time you build an extension. We recommend that you modify your system environment so that you don't have to run *vcvars32.bat* every time. If you look in the *vcvars32.bat* file, you'll see what system environment variables need to be set to make this work.

     [This document is a rough draft. Here we should document how to run Cygwin if you already have MKS installed... basically, change the PATH to put Cygwin at start and unset the ENV variable.]

  You'll use the Windows compiler Visual C++ 5.0. (Version 6.0 may also work).
- As of this writing, TEA isn't compatible with the Macintosh.

For more about Autoconf, see *http://sourceware.cygnus.com/autoconf/*.

The tcl.m4 File

The *tcl.m4* file, which comes with the Tcl Sample Extension, defines the standard Tcl options; these are listed in Appendix A. *tcl.m4* is written in the M4 macro language. Although it's helpful to understand M4, that isn't required.

The easiest way to add options for your extension is by modifying the *configure.in* file, as the next section explains. If your extension has many additional options, you may want to create a separate M4 file.

The *tcl.m4* file (plus any separate M4 files you might create) must be appended to the *aclocal.m4* file before you run **autoconf**. **Autoconf** reads its macros from *aclocal.m4*.

The configure.in File

The *configure.in* file is written in the M4 macro language. The Tcl Sample Extension comes with both *tcl.m4* (which you don't need to edit) and a sample *configure.in* (which you do). As with *tcl.m4*, you can probably get by without understanding the details of M4 and, in this section, we'll show you how.

You'll edit two constructs in *configure.in*: macro calls and variables. A macro call starts with a name in UPPERCASE and has optional argument(s) in a single set of parentheses. For example, here's the AC_INIT macro with the argument exampleA.h:

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

A variable is written *NAME=value*. The name is an UPPERCASE string. It's followed by an equal sign (=) but no spaces. The value can be fairly complex: strings and/or expanded variables, possibly with quotes around them. For instance, the following line sets the variable named FOO to the value *bar*:

```
FOO=bar
```

The next example sets the variable FOO to contain the value of the variable BAR followed by the string xyz, a dot (.) and finally the value of the variable BAZ:

```
FOO=${BAR}xyz.${BAZ}
```

When you edit *configure.in*, your main job will be to replace the abbreviated name of this sample extension, exampleA, with an abbreviation for your extension. For example, if your extension is named Tclbaz, you could edit the variable *exampleA*_LIB_FILE to become *TCLBAZ*_LIB_FILE. Handy tip: use your text editor's search function to find the string "exampleA" and replace it with (for example) "TCLBAZ" or "tclbaz".

You'll need to edit three parts of *configure.in*: one each at the start, middle and end. (Most code in the middle is "boilerplate" that applies to all extensions and doesn't need editing.) Let's look at each edit.

The AC_INIT Macro

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

The AC_INIT macro is required. It verifies the location of the source files for the extension. (It does not test for the existence of a file.) It's important to use AC_INIT because someone can build an extension from a directory outside of the extension's source tree.

The argument to AC_INIT is the pathname to any file that's guaranteed to be part of your extension code and appears only once. Because users may unpack the source code in any directory, this pathname should be relative to the current directory (the directory containing the *configure* file). For instance, if all of the source files for your extension come from the same directory and you want to test for the presence of the file *exampleA.h* in that directory, use AC_INIT(*exampleA*.h). In a bigger extension with multiple directories, you could use a macro call like AC_INIT(foo/bar.c) to refer to *bar.c* in the *foo* subdirectory or AC_INIT(../foo/bar.c) if that file is in a sibling directory (a directory at the same depth in the tree as the current directory).

Extension Name and Version Numbers

```
PACKAGE=exampleA

MAJOR_VERSION=0
MINOR_VERSION=2
PATCHLEVEL=

VERSION=${MAJOR_VERSION}.${MINOR_VERSION}${PATCHLEVEL}
NODOT_VERSION=${MAJOR_VERSION}${MINOR_VERSION}
```

The next section of *configure.in* has a series of variables whose values are often used to build longer strings. For instance, the major and minor version number variables are combined with the patchlevel to make a complete version number. So, if your TclBaz extension is version 2.1, you'd set MAJOR_VERSION to 2 and MINOR_VERSION to 1. If the source code doesn't have a patchlevel, leave the PATCHLEVEL variable with an empty value. (Don't delete unused variables. Leave them as placeholders with an equal sign but no value.)

The PACKAGE is the full name of your package. See the discussion of package names in Chapter 4.

Next come two version number strings that become part of the library name on different systems. The first variable, VERSION, is for systems such as some versions of UNIX that allow multiple dots (.) in their filenames. The second variable, NODOT_VERSION, is for systems like Microsoft Windows that don't use dots.

Miscellaneous Platform-Specific Items

```
case "‘uname -s‘" in
    *win32* | *WIN32* | *CYGWIN_NT*)
        AC_DEFINE(BUILD_exampleA)
```

Near the middle of the sample *configure.in* file is a single macro call that you should change to your extension name. In the call AC_DEFINE (BUILD_exampleA), change the "exampleA". This code sets up function declarations to allow dynamic loading on Microsoft Windows. See the sample *exampleA.h* file for an example.

Shared Libraries and Static Libraries

```
AC_SUBST(exampleA_LIB_FILE)
AC_SUBST(SHLIB_LD_LIBS)
```

The Makefile.in File

The Configure process creates a Makefile that controls the actual building and installation of the extension. (A Makefile is read by the widely used **make** utility. One good source of information is the book *Managing Projects with Make* from O'Reilly & Associates, *http://www.oreilly.com/*.) The extension writer needs to supply *Makefile.in*, which is the template for the Makefile.

[This document is a rough draft. Here we need to add info about Makefiles.]

You'll edit two parts of *Makefile.in*: at the start and the middle. Here's a description of each edit.

Library Name

```
lib_BINARIES = $(exampleA_LIB_FILE)
BINARIES = $(exampleA_LIB_FILE)
```

These two variables set the name of the library you're building. The value for exampleA_LIB_FILE was substituted by the AC_SUBST() macro at the end of the *configure.in* file.

Source File Names

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

The first variable sets the names of all source files (typically, files whose names end with ".c"). The second variable is used when packing the files into a distribution.

Object File Names

```
exampleA_OBJECTS = exampleA.$(OBJEXT) tclexampleA.$(OBJEXT)
exampleA_LIB_FILE = @exampleA_LIB_FILE@
$(exampleA_LIB_FILE)_OBJECTS = $(exampleA_OBJECTS)
OBJECTS = $(exampleA_OBJECTS)
```

The first variable is a list of names of object files. At build time, the $(OBJEXT) macro will hold the extension for object filenames—for instance, ".o" on UNIX and ".obj" on Windows. The second variable's value, `@exampleA_LIB_FILE@,` is substituted by **configure** when it runs; as with all other occurrences of this sample extension's name (`exampleA`), you'll need to change the name to fit your extension.

Header File Names

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

A C header filename typically ends in ".h". List all header files in your extension that need to be installed. Each header file should have a correct pathname. $(srcdir) is the source directory found by the AC_INIT() macro in *configure.in*. If your header files are in some other directory, modify this pathname relative to $(srcdir). So, for example, if your extension has the files *foo.h* and *bar.h* in $(srcdir), you'd use the value *$(srcdir)/foo.h $(srcdir)/bar.h*. Or, if you had one header file *foo.h* in a subdirectory named *include*, you'd use the value *$(srcdir)/include/foo.h*.

You don't need to use a backslash (\) as we do in the sample file. It's a continuation character for variables whose values are longer than one line. This is just an example that shows how to use one.

Other Variables

```
SAMPLE_NEW_VAR=@SAMPLE_NEW_VAR@
```

If you need to set variables in *configure.in* to pass into the Makefile, add a line like the one above for each variable. For instance, your *configure.in* file could define the variable TCL_TOP_DIR by calling AC_SUBST(TCL_TOP_DIR). You'd add a line like this to your *Makefile.in*:

```
TCL_TOP_DIR=@TCL_TOP_DIR@
```

When the **configure** script sees @TCL_TOP_DIR@ above, that string is replaced with the value of TCL_TOP_DIR to create the final *Makefile*.

all Target

```
all: binaries libraries doc
```

The *all* target should always be the first target in the Makefile and have the following dependencies in the order shown above: *binaries*, *libraries* and *doc*. This target simply makes the three sub targets, which implies that the user will have a complete build after this target is called. Since all is the first target, by definition, it's the default target when **make** is called.

This target shouldn't be changed. For instance, if your extension doesn't have documentation, keep *doc* in the list of targets above and leave an empty "doc:" target. The *all* target doesn't need any following commands; it simply makes sure that the three listed targets have been made.

binaries Target

```
binaries: $(LIBRARIES)
```

The *binaries* target builds all platform-specific binaries. Essentially, this target should build any binary that exists in the exec-prefix directory.

This target won't need any following commands. The LIBRARIES are built by the $(exampleA_LIB_FILE) target, later.

libraries Target

The *libraries* target generates any platform-independent files and does any post-processing after the initial binaries have been built. This target should be called after the *binaries* target in case the binaries are needed to build the libraries (e.g., TclX uses the tclx shell to generate tlib library files).

doc Target

```
doc:
        xml2nroff exampleA.xml > exampleA.n
```

```
xml2html exampleA.xml > exampleA.html
```

The *doc* target generates or formats any documentation files, depending on the current platform. This target should be called after the *binaries* target in case the binaries are needed to build the documentation (e.g., generating Windows Help files). The two commands above are just a sample.

Remember that, in a Makefile, each command *must* be indented with a TAB character, not with space characters! This is a common Makefile error.

test Target

The *test* target runs the test suite for the extension. It should set up any necessary environment variables and use the binary on the test suite.

install Target

```
install: all install-binaries install-libraries install-doc
```

The *install* target shouldn't be changed. This target simply calls the three sub targets, which implies that the user will have a completely-installed extension after this target is called. For instance, if your extension doesn't have documentation, keep *install-doc* in the list of targets above and leave an empty "install-doc:" target. The dependency on *all* lets a user run **make install** immediately after **configure**, which minimizes the number of steps needed to build the extension.

install-binaries Target

```
install-binaries: INSTALL_LIB_BINARIES INSTALL_BIN_BINARIES
```

The install-binaries target installs all platform dependent binaries into the specified $(exec-prefix) directory. This should be called only after the binaries target. The actions for the two dependent targets install ".dll", ".so" and executable files into the correct directories. You shouldn't need to modify them.

install-libraries Target

The install-libraries target installs all platform independent libraries into the specified prefix directory. This should be called only after the libraries target.

install-doc Target

```
install-doc:
        $(mkinstalldirs) $(mandir)/man1
        $(mkinstalldirs) $(mandir)/man3
```

```
$(mkinstalldirs) $(mandir)/mann
@for i in $(srcdir)/*.n; \
    do \
    echo "Installing $$i"; \
    rm -f $(mandir)/mann/$$i; \
    $(INSTALL_DATA) $$i $(mandir)/mann/$$i ; \
    chmod 444 $(mandir)/mann/$$i; \
    done
```

The *install-doc* target installs all manuals and related documentation into the appropriate directory. This should be called only after the *doc* target. The example above uses a loop to install Tcl manual pages, in **nroff** format, into three directories.

$(exampleA_LIB_FILE) Target

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

This target and the commands should only need editing to change the extension name. The @MAKE_LIB@ string is a platform-dependent command that's generated and set by **configure**.

Object File Targets

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

tclexampleA.$(OBJEXT): $(srcdir)/tclexampleA.c
        $(COMPILE) -c '@CYGPATH@ $(srcdir)/tclexampleA.c' -o $@
```

The example above has targets and commands to make the object files. For instance, the first target makes the file *exampleA.** (*exampleA.o* on UNIX and *exampleA.obj* on Windows). You should replace those with entries that build each of your object files.

If your header files are in some directory other than $(srcdir), modify the pathname relative to $(srcdir). So, for example, if your extension has the files *foo.c* and *bar.c* in $(srcdir), you'd use the value *$(srcdir)/foo.c $(srcdir)/bar.c*. Or, if you had one source file *foo.c* in a subdirectory named *baz*, you'd use the value *$(srcdir)/baz/foo.c*.

Note that using VPATH and implicit rules (like .c.o:) isn't portable. The method above is the only portable one we've found that allows building from a directory outside of the source tree. (If you have suggestions, please send them to the mailing list. See the Preface.)

clean Target

```
clean:
    -test -z "$(BINARIES)" || rm -f $(BINARIES)
    -rm -f *.o core *.core
    -rm -f *.$(OBJEXT)
    -test -z "$(CLEANFILES)" || rm -f $(CLEANFILES)
```

The *clean* target deletes from the current directory all files that are normally created by building the program. Don't delete the files that record the configuration. Also preserve files that could be made by building but normally aren't because the distribution comes with them.

distclean Target

```
distclean: clean
    -rm -f *.tab.c
    -rm -f Makefile $(CONFIG_CLEAN_FILES)
    -rm -f config.cache config.log stamp-h stamp-h[0-9]*
    -rm -f config.status
```

The *distclean* target deletes all files from the current directory that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, **make distclean** should leave only the files that were in the distribution.

depend Target

The *depend* target generates makefile dependencies and re-generates the *Makefile* from the *Makefile.in*.

Chapter 7
Writing and Running a Test Suite

[This document is a rough draft. Here, we'll explain the new test package. All extension writers should use it.]

For more information, see the section on tests in the Tcl Style Guide, *http://www.scriptics.com/doc/styleGuide.pdf*.

Chapter 8
Documentation

As of August 1999, Tcl reference pages are still written and formatted with **nroff**/**troff** and a special macro package. These are reformatted automatically into HTML, Windows Help, and standard UNIX manual page formats. Soon documents should be written in XML and then translated to other formats.

[This document is a rough draft. Here, we're planning to add information about the document tree, naming the files, and the files' contents.]

Appendix A
Required Configure Switches

An extension that complies with TEA must define all the switches listed here. These switches are implemented automatically by the macros in the *tcl.m4* file supplied with the Tcl Sample Extension.

This is a minimum set; an extension can use any other configure switches it needs. You can add switches by modifying the *configure.in* file or by writing a separate M4 macro file. (If you do, please consider sending your macro to the mailing list!)

1) **--enable-shared (--disable-shared)**

   The *--enable-shared* switch builds the executable as a shared library; *--disable-shared* builds a static executable. The default is *--enable-shared*.

2) **--enable-symbols (--disable-symbols)**

   The *--enable-symbols* switch builds the executable with debug symbols; *--disable-symbols* builds an executable without debug symbols. The default is *--disable-symbols*.

3) **--enable-threads (--disable-threads)**

   The *--enable-threads* switch builds binaries with Tcl threads enabled; *--disable-threads* builds binaries with threads disabled. Currently, the default is *--disable-threads*; this may change when the Tcl thread code is more stable.

4) **--enable-gcc (--disable-gcc)**

   The *--enable-gcc* switch builds the executable using **gcc**; *--disable-gcc* builds the executable using the system default: **cc** on UNIX and **cl** on Windows. The default is to search for the first available compiler that works, defaulting to the system compiler previously specified.

5) **--with-tcl=***DIR*

   The *--with-tcl* switch specifies the build or install directory for Tcl. This is where the system will look for libraries, binaries and the *tclConfig.sh* file. The default location is to look relative to the current location—for example *../../tcl8.1/unix*.