Extension Architecture:

Overview and Requirements

1 Introduction

Downloading from the Internet, configuring, building, and incorporating any more than one extension into any given environment, is a major challenge. Tcl extensions are each different in their own regard, no doubt, and at a minimum they each extend the functionality to the core Tcl language and services. But the configuration properties of each Tcl extension are these days disparate enough that it takes many hours, if not days, to build a complete extended Tcl subsystem that actually works. Attempt this cycle later when Tcl and associated extensions have been upgraded, and your asking for very much the same challenges all over again.

The purpose of this document is to identify the myriad of issues that Tcl users, Tcl extension writers, and Tcl *integrators*, encounter when using, developing and releasing, and incorporating (respectively) Tcl and Tcl extensions. Ultimately the goal of the Tcl developers and Tcl extension developers should be to make Tcl much simpler for their end users—which quite often are mutual Tcl developers themselves.

The time is right for restructuring of the Tcl extension architecture. Given the fact that, for the first time in the development history of Tcl, there will be simultaneous development tracks for multiple versions of Tcl—8.0 and 8.1—there is even more a requirement for such an extension architecture.

1.1 Classes of Tcl Users and Issues They Encounter

To better understand the issues associated with Tcl and Tcl extensions, we can look at the various groups of users who deal with Tcl and Tcl extensions on a regular basis. We look at the Tcl *end user*, the Tcl extension writers, integrators of Tcl into value added products, and packagers of Tcl into larger systems. We shall see that while there are unique some unique problems and issues to each group, there are also many other classes of problems that span these groups as well.

1.1.1 Tcl and Tcl Extension End Users

Simply stated, end users of Tcl and Tcl extensions are our *customers*. Such end users have varying level of sophistication and expertise and thus what is an issue or problem associated with Tcl and extensions for one user may not be an issue or problem for another end user. In addition some end users may themselves be developers who understand how to build and manage Tcl and extension sources, while others may simply want to use Tcl immediately after downloading and installing. Recognizing the issues and problems encountered by our customers will help us better architect a new strategy for deliver Tcl and Tcl extensions.

In general **Tcl end users find it difficult to download and install an extension into an existing Tcl installation**. The simple process of download, install, and run, of Tcl extensions simply does not exist. Quite often extensions are delivered in the form of sources and most Tcl users don't want to have to build (e.g., type a variation of *configure* and *make*) an extension to use it. Requiring them to do so implies the user to have a C compiler environment and possibly knowledge of how their installed Tcl environment was previously built.

Installations of Tcl and Tcl extensions are difficult to manage. During the compile stage Tcl and Tcl extensions incorporate a path describing the location of their respective script libraries. Environment variables equivalent to *TCL_LIBRARY* need to be used to override this compiled–in value. Furthermore, network installations of Tcl are hard to manage in a site where. the value of *–exec-prefix* (and *–prefix*) may be different when building an extension to where an extension is actually installed. For example, it is not uncommon for users to have to configure and build Tcl and Tcl extensions with a interim value for *–exec-prefix* set to */home/joe/tcl/solaris-sparc/bin*, and install Tcl using *–exec-prefix* set to */usr/local/bin*.

Upgrading Tcl and extensions is problematic. When upgrading a Tcl installation, Tcl extension may need to be upgraded as well. Or it might be the case that in order to upgrade a Tcl extension, the Tcl core needs to be upgraded. Users with stable environments are unwilling to take such drastic steps. Ultimately the whole upgrading issues becomes chaotic if one extension or another have not yet released a version of their extension to match a new version of Tcl, leaving the user with the dilemma of least common denominator environment.

A Tcl installation tree does not support building against multiply installed Tcl version. Binary files (Tcl shells, shared libraries, static archives) are versioned, but the *include* directory does not support multiple versions of the Tcl header file; the same is true for the *tclConfig.sh* file which only represents a single configuration. In both cases, only the most recent installation instance of Tcl is represented by these files. For users who use Tcl as a development platform and want to compile against header files in the *include* directory and link against those libraries in the *lib* directory, this poses a problem when different versions of these files are required.

The fact that Tcl and Tcl extensions use **varying form of documentation** makes Tcl harder to learn and use. Users require *nroff* for UNIX, WinHelp for Windows, and perhaps HTML for Mac. While some tools exist in the Tcl core to go from UNIX *nroff* to the latter, extensions authors do not employ them and often have their own formats for UNIX manual pages or HTML documentation. Documentation includes demonstration or example programs as well; some extensions are really good at providing complete applications that show off their feature set, but others are either sloppy, incomplete, or completely without them.

Finally, **Tcl users do not have a single one–stop resource for locating prebuilt Tcl extensions**. Scriptics does provide the *Tcl Resource Center* as a repository for extensions and uploading new extensions to this location is fairly easy. But extensions come in various shapes and sizes—some are Tcl only extensions, but most these days are written in C and require that they be downloaded and built. Overall, the Tcl community needs a better way to distribute Tcl extensions in the form of easily installable entities that just work *out of the box*.

1.1.2 Tcl Extension Authors

Let us next turn our attention to Tcl extension authors.

Over time experienced extension writers have developed special skills to design, implement, and distribute their respective extensions. While this continues to work for the experienced elite of extension authors, there exists a steep learning curve for new extension authors wanting to write their own extensions or ones they wish to distribute. There is no basis or template new extension authors can use to begin their work and reduce this learning curve. Cross platform issues are plentiful and mastering them all takes significant effort. Finally writing documentation for an extension—even ones that have been around a long time—is challenging when having to fulfill many different documentation formats: UNIX *nroff* manual pages, WinHelp, and HTML.

Upgrading to new versions of Tcl is painful. For an extension author to upgrade their work to newer versions of Tcl, this sometimes involves modifying their code to changes or additions to Tcl internal APIs and sometimes even Tcl public APIs. Tcl public and internal APIs change too often and API signatures that change over time cause extensions relying upon them to be brittle. While extensions should not use Tcl internal APIs it is sometimes inevitable for those extension that greatly depend on internal structures (for example, CTRL+C support in TclX) and ultimately find themselves being compatible with only the version they were built against.

Programming languages other than C, such as C++ and Java, need better support. Extension development traditionally began in C, because the Tcl core was developed in C. C++ is now very popular for building Tcl extensions, particularly on platforms where it is the language of choice for non–Tcl development (e.g., *MFC* on Windows). As such, Tcl and Tcl extensions need to deal with this issue head on. For example, there is little in the way of examples on how to develop a Tcl extension in C++ and over time most users discovered and dealt with issues such as *static class methods* and name mangling on their own. But because there is no template example, these types of issues are not widely known. The Tcl header files are guilty of using C++ keyword (e.g., *string* if STL is used) sometimes requiring users to edit these headers when using a compiler that can't deal with this.

Java also poses interesting opportunities for extension developers. While *TclBlend*, exists there is little in the way of support for its facilities in the core. Tcl users and extension developers must rely upon an older version of the Tcl core in order to create Java based extensions.

There are few Tcl–script only Tcl extensions. Such extensions are easier to write, distribute, and maintain. But there is little support in the Tcl core for aiding in this endeavor. The *tclIndex* and *pkgIndex.tcl* mechanisms often conflict with each other creating brittle runtime environments. Tools for building indexes for Tcl–script only extensions exist (e.g., *pkg_mkIndex*), but they have not proven to work (until maybe recently) and Tcl users are left to create *pkgIndex.tcl* files by hand. As such, extensions such as TclX have their own command indexing schemes that are very different than those built into the core either because of history or resistance to move to something standard. Finally, while *namespace*s do exist, there is still a potential for name collision; there is no central repository for registering *namespace* names for publicly distributed extensions.

Finally, assuming a Tcl extension author has successfully written an extension, there comes a point when it's time to release the product. Unfortunately **there is no standard packaging utility to easily distribute extensions other than in source form**. In addition the expectation of end–users vary from platform to platform: UNIX users are accustomed to receiving source distributions; Windows and Mac users are more likely to accept native installation applications. Some other issues related to installation of an extension (but not limited to) include ensuring that the target Tcl installation hierarchy is version compatible, copying all required to the correct places in the Tcl installation hierarchy, etc.

1.1.3 Tcl Extension *Integrators*

We refer to integrators as people who want to integrate Tcl and/or Tcl extensions into as part of a larger product. We may also classify integrators as *advanced* Tcl users, so therefore man of the same issue described in sections *1.1.1* and *1.1.2* may equally apply to these folks as well. Integration may take the form of, but not limited to:

- using Tcl and Tcl extensions as tools to perform product related tasks
- embedding the Tcl interpreter and Tcl extensions into an application
- adding their own proprietary extensions or value to the Tcl core.

An example of one such integrator is Scriptics development team that has built *TclPro* utilizing Tcl and Tcl extensions in the ways described above.

Extensions don't inherit information from the configuration process of Tcl via *tclConfig.sh* on UNIX or *makefile.vc* on Windows. Compiler switches configured by Tcl on UNIX are lost. Extensions are not built with the same configuration as the Tcl core. An integrated product needs to have a consistent build configuration and extensions that have poor configuration schemes create unreliable runtime environments.

Building static Tcl and static Tcl extension variants is difficult. It is interesting to not that several years ago, Tcl and Tcl extensions were all statically linked together. Extending a Tcl shell simply meant adding the extension's *Init routine to your Tcl_AppInit(..)* and augmented the link line for your Tcl shell to include the new extension. With the introduction of the *load* command and the package mechanism, support for building completely static Tcl shells with extensions almost is non–existent—even in the Tcl core. Issues include those related to *makefiles*, the right way to initialize an extension (especially when taking sub–interpretters into consideration), and priming the package mechanism for a *static package*.

During development of large complex projects, **building a debug only environment of the Tcl core and Tcl extensions is often required but hard to do**. In general, during development, if Tcl is built for debugging, it is probably desirable to have the remaining Tcl extensions also built for debugging. In fact on Windows, it is unfortunate, but you cannot have a mixed environment of debug and non–debug modules—it's debug or all non–debug. Extensions provide little or no support for adding the necessary compiler flags for debugging and do not link with the correct Tcl libraries (dynamic or otherwise) to create a well behaved development environment.

Building an integrated product of Tcl, Tcl extensions, and value add tools is a major task. Most end users build Tcl and Tcl extensions probably only as often as required to keep up with new versions or site configuration changes. Integrators on the other hand, need to build many variants of Tcl and extensions (dynamic, static, debug, non–debug, etc.) on a regular basis. Automation of building such a product is extremely important—especially towards the release phase of a project.

Lastly, as stated earlier, upgrading of extensions, C++ and Java issues, disparate extension documentation formats, and the bevy of other issues that affect Tcl extension writers and end users of Tcl also relate to the Tcl integrator. For the integrator–developer these problems compound themselves and directly effect project schedules and product development.

1.1.4 Tcl Packagers

We refer to *packagers* as people or organizations that include Tcl and Tcl extensions as part of their complete product offering. Packagers are different from integrators in that they are not trying to add value to Tcl or Tcl extensions. They may not even be supporting Tcl, but simply packaging it in. Examples of packagers include Red Hat Linux who package Tcl and Tcl extensions into *Red Hat Linux* in the form of RPMs.

For this class of user, it must be easy for a third party (the "packager") to download a source distribution, build it on a particular platform, and create a built distribution for that platform which can then be trivially used by less sophisticated users of that platform. Similar to integrators, reproducibility and reliability of source distributions of Tcl and extensions is very important.

2 Requirements

From the discussions of problems and issues faced by the various classes of Tcl users, we can group requirements for the Tcl Extension Architecture into a few categories: build configuration, documentation, testing, distribution and installation. For each of these areas we will specify a set of requirements that will comprise the Tcl Extension Architecture. Each of these areas and their requirements are given below. Further in this document we will propose some solutions and changes to the Tcl core and Tcl extensions to satisfy these requirements.

Tcl and Tcl Extension Build Configuration

- Keep the configuration, build, test, and install of extensions simple in the most common case. This includes *configure* and *make install* on UNIX and simply *nmake install* on Windows. The default behaviors for all switches should be well understood by Tcl and all extensions.
- All extensions should support the same *configure* script switches on UNIX. Where Windows is supported, *Makefile* targets and macros for compiling, linking, installing, and testing should be similar on UNIX and Windows.
- Extensions should rely more upon *tclConfig.sh* for the commonly used compiler and linker flags.
- Reproducibility of configuration, installation, and testing should exist across all extensions. You should be able to easily write a script that builds any combination of extensions
- While no extension architecture framework can help with platform specific porting issues for an extension, there needs to be a better way to get the simplest of extensions—be it extensions that don't rely on platform specific C APIs or Tcl script only extensions—across platforms very easily.
- The Tcl Extension Architecture should facilitate developing extensions for Windows with an adequate template *Makefile* suitable for MS Visual C++.
- If Tcl APIs retain their signatures and new APIs are added to the Tcl core to give new functionality, then extensions can live in a heterogeneous world of Tcl and Tcl extension versions.
- The Windows *Makefile.vc* files for Tcl and Tcl extensions should take extra care to link to the correct instance of the Microsoft C Runtime library when building dynamically linked libraries.
- An issues related to creating binary distributions: We need to break the notion that the site where Tcl or Tcl extension are built is the site where Tcl or Tcl extensions will be installed. In other words, the use of *–exec-prefix* and *–prefix* need to be adjusted for this fact.

Tcl and Tcl Extension Documentation

- There should be one document source that Tcl and Tcl extension developers agree upon.
- This document source should produce clean and consistent end user documentation for all platforms: *nroff* manual pages on UNIX, WinHelp on Windows, and HTML for Mac.

Tcl and Tcl Extension Testing

- Every extension should follow a simple framework for testing of a built extension. White box tests as well as black box tests should be included where appropriate.
- Test directories should be similar to the Tcl test framework.
- White box and black box tests should be provided by extensions where possible and appropriate
- Tests that create intermediate files during the testing process should clean up after execution.
- Tests should provide adequate results that users can use to troubleshoot problems effectively.

Distribution and Installation of Tcl Extensions

- Make it easy for Tcl users to get up and running as quickly as possible with Tcl and Tcl extensions—no build step required on the part of the Tcl user, who simply wants to use Tcl without having to build it.

- The above requirement implies that we need to make binaries for Tcl and Tcl extensions more readily available so Tcl user's can *download and go*.
- If binary distributions are to work, we need to remove—or make less visible—the fact that environment variables for Tcl and Tcl extensions exist. Leave support for environment variables specifying script library locations for use during development. We need to reduce the visibility of the *TCL_LIBRARY* environment variable and other such mechanisms to the casual Tcl user.
- Tcl extensions need to be less dependant, and more resilient, on the version of the Tcl core they require. At the least, the extension needs to check at runtime, when it is loaded, whether it is compatible with the version of the Tcl core that loaded it.
- File layout of installed hierarchy for Tcl and Tcl extensions should support multiple versions of Tcl.

3 Tcl Extension Configuration

The requirements set forth for configuration of Tcl and Tcl extensions are geared mainly toward development issues. Many are based on the collective experience of groups of individuals alike: Tcl end users, Tcl developers, and Tcl extension developers. Some issues are platform specific, others are general subtle issues relating to dynamic linking, static linking, multiple versions of Tcl, etc.

3.1 Some Platform Specific Issues

The purpose of this section is to briefly mention some platform specific issues and also about. Presented below are issues regarding the various ways Tcl and Tcl extensions are configured and built today on both UNIX and Windows. Some these issues are particularly problematic for most all of the class of users discussed earlier in this document. As we think about the Tcl Extension Architecture, we should be factoring these issues into our solution.

3.1.1 Configuring and Building for UNIX

Because Tcl and Tcl extensions are traditionally developed on UNIX and given the fact that UNIX tools are fairly robust and powerful, building a Tcl subsystem is fairly well understood among the masses. The Tcl *configure* mechanism alone is fairly good about determining individual UNIX operating system/host/compiler flags on its own. Other issues include library switches to compiler drivers and linkers.

Over time, however, extensions have developed their own personalities and ways of configuring and building. It is sometimes possible for a Tcl extension to be built incorrectly because it did not inherit some special compiler switches determined during the *configure* step of Tcl. We recently saw this at Scriptics when we built TclX and Expect for Irix; both extensions had missed the *–n32* flag required by the SGI compiler.

One solution for this is for extensions to rely entirely upon the *tclConfig.sh* shell script generated during the *configure* step of Tcl itself. This script should be the central repository for as many compiler and linker flags as possible. Tcl extensions should augment the compiler and linker lines only those flags they need that do not already exist.

3.1.2 Configuring and Building for Windows

Sad but true, it is often the case that Windows follows an implementation of an extension on UNIX—or not in the case of Expect. As UNIX is generally the reference platform for most extension developers, Windows versions of extensions often follow long after the original UNIX implementation; with such work often performed by someone other than the original extension author. A few reasons why certain extensions have this characteristic are:

- As in the case of Expect the functionality is difficult, if not impossible to implement on Windows.
- The extension author does not have the resources or expertise with the Windows host.
- It is too difficult to create a distribution for yet another platform.

The latter of these factors should not be an impediment for other developers to take the extension and make it available on the Windows host. However, there is little in the way of support in the core of Tcl for moving an extension to Windows—assuming the functionality is technically feasible. The lack of a mechanism similar to the UNIX *configure*/*make* can and is definitely an impediment. Our Tcl extension distribution and installation strategy should address deploying extensions on Windows head on!

As Tcl continues to become popular on the Windows, this pattern will reverse and we will begin to see extensions that originate on the Windows platform. APIs for such things as ActiveX, COM, and MAPI are likely candidates for extensions originating on Windows.

In addition there are subtle differences in the way dynamic libraries and debug libraries are built on Windows (see *3.1.3 Building Dynamically Linked Tcl Extensions* and *3.1.5 Building Debug vs. Non–debug Extension Libraries*). As well, static Tcl shells that have extensions statically linked have a different set of issues than the dynamic case (see *3.1.4 Building Statically Linked Tcl Shells — "Big" Tcl Shells*).

It is also safe to say that generally there is only one compiler in use today, Microsoft Visual C++. Most of the Tcl extension architecture will be centered around this compiler platform.

There is the GNU C compiler available based on the CYGWIN library. Support for Tcl in a CYGWIN environment is derived from the UNIX *configure*/*make* mechanism and any extension architecture implementation that is made will be readily inherited. While CYGWIN may be used in a development environment where UNIX and Windows are mixed, it is not common where Microsoft Visual C++ is the predominant Windows development environment. Most certainly it is not used as a delivery vehicle for Windows applications to end users.

3.1.3 Building Dynamically Linked Tcl Extensions

In general Tcl and Tcl extensions are built dynamically. This reduces code–bloat and makes it easier to maintain and manage installations of Tcl and Tcl extensions. In any single Tcl subsystem environment, either everything is built dynamically or everything is built statically. (The issues associated with building Tcl and Tcl extensions statically, usually uncommon, is discussed in *3.1.4 Building Statically Linked Tcl Shells — "Big" Tcl Shells*.) Rare is the case that environments are mixed between dynamic and static and is in fact problematic on UNIX and downright impossible on Windows.

Closely related to dynamically built Tcl extensions is the topic of the work–in–progress of the new *stubs* library discussed in *3.2 The Stub Library*. This work will enable the need to not to have to link directly to Tcl shared library on UNIX or to a Tcl DLL export library on Windows.

With the proposal described in *3.1.1* to do away with *--enable-shared* and *--enable-symbols* building dynamic link libraries of either Tcl or Tcl extensions is now fairly straight forward.

As long as care is taken to ensure that everything in a Tcl subsystem is built dynamically then everything should just work. In other words the Visual C++ compiler switches */MD* needs to be used when build code for linking and */MDd* when code for dynamic linking in a debug environment.

3.1.4 Building Statically Linked Tcl Shells –– "Big" Tcl Shells

A framework for building a statically linked Tcl shell consisting of any combination of Tcl extensions. is currently missing The framework should require very little from the user which may include adding code to their *Tcl_AppInit(…)* as described in below, but easily add the necessary static archive entries for a respective extension to a *Makefile*.

It turns out that the *Tcl_AppInit* procedure for a statically built Tcl shell that contains any number extensions is fairly simple to create. Generally all that is required are *Tcl_StaticPackage(…)* statements for each extension that is statically linked into the Tcl shell. Providing the value of *NULL* for the pointer to the interpreter to *Tcl_StaticPackage(…)* arranges for the statically linked packaged to be loaded whenever *package require* is used for that extension—even for a sub–interp.

For each extension that is statically linked into a Tcl shell, the equivalent of a *package ifneeded* statement is required; the associated is script performs a *load {}* which effectively synthesizes the loading of the statically linked extension and ultimately calls its *_Init* routine. Such a file should exist in a place where the statically linked shell can access it when needed. Unfortunately the location of this file needs to be in a place that can be seen by the static package mechanism and not conflict with the dynamic package mechanism.

Alternatively, the call to *Tcl_StaticPackage(…)* for each statically linked extension needs to handle the synthesis of this *package ifneeded* process. Such functionality does not currently exist in *Tcl_StaticPackage(…)* today and will need to be made if statically linked packages are to work seamlessly.

3.1.5 Building Debug vs. Non–debug Extension Libraries

Similar to dynamic and static builds, generally if a Tcl shell is built for debugging, it is likely that other extensions want to be built for debugging as well. Tcl and Tk 8.0.3 added a special tag to the target binaries to denote that they contain debug information. The target names includes the "g" character on UNIX and "d" character on Windows. So for example, *libtcl80g.a* contains debugging symbols of Tcl.

A new variable in the *tclConfig.sh*, *TCL_DBGX*, now exists and will contain the value "g" if Tcl is built with *--enable-symbols*. Currently UNIX *configure* scripts for Tcl and Tk appy the tag of "g" to dynamic shared objects and static archives that are built with *--enable-symbols*. The proposal to do away with the *–enable-** switches won't change this, but when *make* is invoked with the *DEBUG* macro set to 0 (or 1), the associated targets will need to deal without applying the *TCL_DBGX* macro to generated libraries (or apply).

On Windows, it is a foregone conclusion that s system of modules cannot be built with mixed debug and non–debug. This is because any module built for debug (or non-debug) must link with the Microsoft C runtime library built for debug, *MSVCRTD.LIB* (or non–debug, *MSVCRT.LIB*). A system of modules loaded at runtime in this manner, result in having multiple instances of the C runtime library—which include multiple instances of the memory heap and other C runtime global variables such as *errno.*. Similarly a static library built for debug (or non–debugging) must link with the static Microsoft C runtime library built for debug *LIBCD.LIB* (or non–debug, *LIBC.LIB*).

3.2 The Stub Library

3.2.1 Requirements

- Extensions should be loadable into multiple versions of Tcl without being recompiled or relinked.
- Extensions should be loadable by statically linked Tcl interpreters.
- Extensions should be able to export their own interfaces using the same mechanism as Tcl.

3.2.2 Description

The Tcl stubs mechanism defines a way to dynamically bind extensions to a particular Tcl implementation at run time. This is accomplished by exporting function tables that define a particular interface. All access to the interface in question is done through offsets into the function table so there are no direct references to any of the library's symbols.

3.2.3 Using the Stub Library

In order to use the stubs interface to Tcl or Tk, an extension must do the following:

1. define USE_TCL_STUBS
2. link against -ltclstub (a static library) instead of -ltcl
3. call Tcl_Required() as the first statement

This will initialize the Tcl function vector and set up macros so that every call ends up going through this table. The resulting library will have no linker dependencies on the Tcl library. Everything is resolved at initialization time when Tcl_Required() is called.

3.2.4 Defining a New Stub Library

Extensions that want to define a stub interface of their own need to do the following:

1. Define a foo.decls file that describes the interface
2. Run the genStubs.tcl script to create the stub files
3. Define a Foo_Required() function that intializes the stubs
4. Compile and link the stub files into a static library
5. Link the fooStubInit.c file into the extension

6. Deliver the fooDecls.h and foostub.lib files to the user

3.3 Tcl Extension Versioning Issues

With the introduction of the *stubs* library (*3.2 The Stub Library*) it should now be possible for extensions to distance themselves from keeping in sync with Tcl versions as they are released. Now that it is possible for extensions to query the Tcl core for its version, extensions can be written to be *backward compatible* through API interfaces provided via the stubs library.

In particular version numbers need to be isolated from versions of Tcl. For example, TclX now need to have a version number consistent with that of Tcl (Tcl 8.0.5, Tclx8.0.5).

3.4 Configuring and Making

3.4.1 Simplifying tclConfig.sh

It has also been suggested that the *tclConfig.sh* file has outgrown its purpose in life, containing too many variables and indirections making it harder to use over time. A long term alternative to the *tclConfig.sh* file is for Tcl to be able to report at runtime, the way in which it was built. Such a report may include the Tcl version, the compiler switches, the linker switches, whether it is built dynamically or statically, whether it is built with debugging symbols or not, etc.. Extensions that are to be built against and work against a particular instance of Tcl can interrogate the Tcl runtime for all salient information required to build itself.

Proposal to Do Away with the —*enable–shared* and —*enable-symbols* Flags of Tcl's UNIX *configure*: To achieve the uniformity discussed above, we propose to remove the flags *--enable-shared*, *--enable-symbols*, *--disable-shared* from Tcl's UNIX *configure* script. This will basically push the decision of building either dynamic or static and either debug or non-debug down into the *Makefile*. Once we do this, the *tclConfig.sh* file will contain all the necessary variables to build either static, dynamic, debug, or non-debug. In other words *tclConfig.sh* will contain the union of all variables that would individually be generated today if you use the associated *–enable-?* flag.

As part of this change, we propose that each variant is built to a separate target directory. So dynamic build would go to a directory named *dynamic*, dynamic/debug will go to a directory named *dynamic-debug*, etc.

The result of this change finally is that on UNIX you *configure* once and from one generated *Makefile* build any combination of either static or dynamic and either debug or non-debug. It does, however, have the added disadvantage that the *Makefile* is a bit more complicated and must include implementations of targets that conditionally test for values of macros selected by the user on which variant to build (see *3.5.3 Makefile Targets for both UNIX and Windows*).

3.4.2 File Layout of Installed Tcl

Currently the installed Tcl tree includes the binaries (executables, shared libraries, and static archives), script libraries, header files, and some configuration files (on UNIX, *tclConfig.sh*). Except for the files in the *include* directory and the case of the *tclConfig.sh* (on UNX), multiple versions of all other files (binaries and script libraries) can live heterogeneously. This is because the naming of these files include the version number from which the particular file was generated (binary) or contributed from (script libraries).

So therefore users who build extensions against a Tcl installation tree have access to only the version of the headers and *tclConfig.sh* that were installed most recently. In other words, even though the installation tree may contain multiple versions of the Tcl runtime, there is effectively only one version of Tcl files used for building against.

Even though this may be problematic, we propose **not** to change this structure at this time. In general if an extension is to be built from an installed version of Tcl, it is likely that the version of the header file in the *include* directory will more than likely suffice. If you consider that installing into a Tcl installation tree is chronologically oriented (maybe), then a user is likely to find the most recent version of the Tcl header files in the *include* directory. Regarding the *tclConfig.sh* file, the proposal to make this file configuration inspecific (do away with —*enable–symbols* and —*enable–shared*) also will likely represent configuration information of the most recent installed version of Tcl.

So, as it is today, if a user wants to build an extension against an alternate version of headers and configuration files than those in the installed Tcl tree, they will have to resort to a source distribution of Tcl.

3.5 configure *Flags,* tclConfig.sh *Variabls, and* Makefile *Targets and Macros*

Configure switches and .*Makefile* targets today are disparate in nature and it is difficult, if not annoying, to have to figure out and deal with these disparities. For example, Tk uses the switch --*with-tcl*; while Expect provides the same functionality with the flag --*with-tclconfig*.

An ideal goal is being able to reproduce the download, configure, and build cycle on a regular basis for Tcl and all other extensions. For example, given that Scriptics currently hosts the CVS repositories for Tcl, Tk, TclX, Expect, and [incr Tcl], we should be able to synthesize the download via CVS checkouts, configure, and build all extensions in various configurations on a regular basis.

Given the new proposal to do away with the *–enable-** flags, we summarize below the list of *configure* flags extended by Tcl, variables generated in *tclConfig.sh*, and *Makefile* targets uniform across UNIX and Windws.

3.5.1 configure Flag

Flags to *configure* remain as before but are fewer with the proposal to remove --*enable-shared* and --*enable-symbols*.

- --*enable-gcc*: Specifies that the configuration process should look for and use GNU *gcc* as the compiler of choice. Not using this flag indicates configuration should use the native compiler, often *cc*,.
- --*with-tcl=<dir>*: Specifies the location of the *Tcl* configuration file, *tclConfig.sh*.
- --*with-tk=<dir>*: Specifies the location of the *Tk* configuration file, *tclConfig.sh*. Extensions that need to compile and link against Tk header files may use this flag.

3.5.2 *tclConfig.sh* Variables

Flags that exist today to represent a singular build of either dynamic or static, will need to be duplicated for both dynamic and static. To represent the complete set of configurations (dynamic, dynamic/debug, static, and static/debug), not many variables need to change. In fact most every flag required for multiple configurations currently exists. There are some variables such as *TCL_LIB_FILE*, that need to be replicated. The complete set of *tclConfig.sh* flags include:

- *TCL_DYNAMIC_LIB_FILE*: This is the name of the archive that is created when
- TODO: To be filled in after implementing new *configure*/*makefile* reorganization on UNIX.

3.5.3 *Makefile* Targets for both UNIX and Windows

Where before a singular *makefile* only built either static or dynamic will now need to build both. This implies almost twice the number of targets. UNIX only built the variant configured during the *configure* process using (or not using) the --*enable-shared* and --*enable-symbols* flags. On Windows, only shared (dynamic) versions of the Tcl libraries were built so it was only possible to build dynamic extensions.

Building for either dynamic or static will be selected based on a macro at the top of the *Makefile*, *DYNAMIC*. *DYNAMIC* will take the default of 1 and can either be changed in the file or on the command line. Other *Makefile* variables below this macro definition will conditionally switch their contents based on the boolean value of based on the boolean value of *DYNAMIC.*

Building for either debug or non-debug will be selected based on a macro at the top of the Makefile, *DEBUG*. *DEBUG* will take the default of 0 and can either be changed in the file or on the command line. Variables below this macro definition that need to conditionally switch their contents based on the boolean value of *DEBUG* and do not include the flags for non-debugging (if *DEBUG* set to 0, such as “” for *CFLAGS*) or debugging if *DEBUG* set to 1, such as “-g” for *CFLAGS*).

The actual implementation of switching compiler and linker flags based on the boolean values will vary between UNIX and Windows, because simple *make* on UNIX (not GNU *make*) does not support conditional testing as *nmake* on Windows does.

The minimum set of targets a Tcl extension should support are:

- `all:` Builds the configuration set by the values of the *DEBUG* and *DYNAMIC* macors

- `libraries:` Builds the extension specific script libraries.
- `man:` Builds the extension specific documentation files.
- `install: install-binaries install-libraries install-man`
- `install-libraries:` Installs the extension specific script libraries in the location given by —*prefix* during *configure* process.
- `install-man:` Installs the extension specific documentation files in the location given by —*prefix* during *configure* process.
- `test:` Performs the test procedures for the built extension (see *5 Testing Strategies*).
- `dist:` Creates a source distribution for the Tcl extension.
- `setup:` Creates a binary distribution for the Tcl extension (see *6 Distribution and Installation*).
- `clean:` Removes all derived objects in the current build area.
- `what:` Echo to standard output all the text here—essentially a usage string for this *Makefile*.

An example of the implementation of the *all* target is:

```
all:
        @if test "$(DYNAMIC)" = "1" -a "$(DEBUG)" = "0"; then \
            OBJDIR=$(DYNAMIC_OBJDIR); \
            TCL_LIB_FILE=$(TCL_DYNAMIC_LIB_FILE); \
            CC_SWITCHES='-o $$@ $(CC_DYNAMIC_SWITCHES)'; \
        else if test "$(DYNAMIC)" = "1" -a "$(DEBUG)" = "1"; then \
            OBJDIR=$(DYNAMIC_DEBUG_OBJDIR); \
            TCL_LIB_FILE=$(TCL_DYNAMIC_DEBUG_LIB_FILE); \
            CC_SWITCHES='-o $$@ $(CC_DYNAMIC_DEBUG_SWITCHES)'; \
        else if test "$(DYNAMIC)" = "0" -a "$(DEBUG)" = "0"; then \
            OBJDIR=$(STATIC_OBJDIR); \
            TCL_LIB_FILE=$(TCL_STATIC_LIB_FILE); \
            CC_SWITCHES='-o $$@ $(CC_STATIC_SWITCHES)'; \
        else if test "$(DYNAMIC)" = "0" -a "$(DEBUG)" = "1"; then \
            OBJDIR=$(STATIC_DEBUG_OBJDIR); \
            TCL_LIB_FILE=$(TCL_STATIC_DEBUG_LIB_FILE); \
            CC_SWITCHES='-o $$@ $(CC_STATIC_DEBUG_SWITCHES)'; \
        else \
            echo "Error: Must specify values of DYNAMIC and DEBUG!"; \
            exit; \
        fi; fi; fi; fi; \
        if test ! -d $$OBJDIR; then mkdir $$OBJDIR; fi; \
        TCLSH=$$OBJDIR/tclsh; \
        $(MAKE) $(MAKEFLAGS) \
                OBJDIR=$$OBJDIR \
                TCL_LIB_FILE=$$TCL_LIB_FILE \
                CC_SWITCHES="$$CC_SWITCHES" \
                TCLSH=$$OBJDIR/tclsh \
                $$TCL_LIB_FILE $$TCL_STUB_LIB_FILE $$TCLSH
```

3.6 Tcl–script Only Extensions

Tcl–script only extensions are easier to write and deploy because they require no C code and thus no configuration and compiler to build them. But the package mechanism for such extensions is currently broken or weak in some regards. As part of the Tcl Extension Architecture we should address any and all issues with this mechanism and even consider revamping it if necessary.

An issue for Tcl–script only extensions is how do you define a package that spans multiple files? The problem is that the *namespace* command may be in one file but the other file (which relies on state in the *namespace eval*) gets called first. The result can cause Tcl errors that are hard to track down. A solution to this issue is to use the *-direct* fag of *pkg_mkInd*. Doing so will arrange for the entire package to be loaded immediately upon *package require* instead of delaying loading until actual use of one of the commands.

Proposal to fix *pkg_mkIndex*: In the short term, we propose here to change the *pkg_mkIndex* command to make the –*direct* flag the default. Doing this will have the unfortunate performance side–affect that all files of a package will be sourced before any routine in the package is used. We also propose to add the flag –indirect to the *pkg_mkIndex* command. A Tcl–script only developer who is concerned about performance can use this flag and can arrange for the core file containing the *namespace eval* and the remaining package source files to all be sourced when any method of a *namespace* is invoked.

Some questions raised by Jeffrey Hobbs (but not necessarily restricted to Tcl only extensions):

1. Problems with inter-related package requires
2. How does one deal with the namespace properly (conflict aversion, import/export)
3. Using packages in slave interps
4. Why do I have to always build pkgIndex.tcl by hand to get it to work right?
5. "The whole auto_path/tclIndex/pkgIndex.tcl/tcl_library/...thing isn't working/doesn't make sense..."

3.7 Environment Variables and Windows Registry Entries

Moving forward, Scriptics proposes to deprecate the environment variables, TCL_LIBRARY and TK_LIBRARY, etc. for Tcl installations. These environment variables may be necessary for development where intermediate Tcl shells are not *installed* and need to be told where the Tcl script library is. This basically involves removing references to these environment variables from documentation where an *end–user* may not see this information.

Similarly, on Windows, we should deprecate the registry entry for *tcl_PkgPath*. These environment variables have in the past been problematic and prone to error.

An issue, however, with attempting to remove them is the fact that the Tcl test suite requires these environment variables today to specify the location of the Tcl script library; because the directory from which the binary resides may not be anywhere relative to the location of the Tcl script library search mechanism.

More extensions should begin to support searching for their script library relative to the installed Tcl library; i.e., relative to the value of the *tcl_library* Tcl variable. Doing so will ensure that all extensions are installed in a clean and consistent environment. If however an extensions script library is not installed relative to the Tcl library, then the value of the extension library variable, for example *tclx_library*, should be used. A user, or script, could set *tclx_library* to the correct location of the TclX script library and then proceed to *package require Tclx*.

To assist in located an extensions script library, extension initialization routines should use the new Tcl library routine *tcl_findLibrary* located in Tcl's *init.tcl* file to locate their library initialization script file.

4 Documentation Strategy

For many of the reasons outlined in *1.1.1 Tcl and Tcl Extension End Users* and *1.1.2 Tcl Extension Authors*, there is a definite need for a unified documentation strategy for the Tcl core and Tcl extensions. Currently there are too many maintenance issues associated with UNIX *nroff* manual pages and the tools associated with creating the associated WinHelp and HTML equivalents.

It is our goal at Scriptics to one day create a unified documentation strategy for at least Tcl and Tk and suggest that extensions employ it as well. XML has been proposed as the source form for such a strategy. Moving to an XML will allow Tcl to easily generate UNIX *nroff*, HTML, and Windows RTF help formats. Adoption of XML cannot happen overnight, however. But to reach this goal it is probably best for the Tcl core developers and Tcl extension developers to agree upon a standardized DTD, that defines the structure of the source XML, in the short term, while planning the work for actually moving existing *nroff* based documentation to XML.

Richard Hipp has put together the beginnings of XML based Tcl documentation. Fore more information see http://www.hwaci.com/sw/tcldoc/. A preliminary DTD has been written by Joe English and is available at: http://www.egroups.com/list/tcl-xmldoc/ and is included below for reference:

```
<!--
    $Id: tmml.dtd,v 1.3 1999/02/17 01:57:54 jenglish Exp $
```

```
    Author:     Joe English, < jo-@flightlab.com >
    Created:    8 Feb 1999

    XML DTD for TMML, version 0.2.

    Notes:
        Some elements have a #FIXED attribute named NOTATION.
        This is a temporary place-holder to indicate that they
        have a special semantic interpretation (for example,
        cross-references).  In the future these will be replaced
        with a general-purpose architecture that can be automatically
        processed.

        "%%%" is a "FIXME indicator".

    Usage:

        For a single manual page:
            <!DOCTYPE MANPAGE SYSTEM "tmml.dtd">

        For a complete reference manual:
            <!DOCTYPE TCLDOC SYSTEM "tmml.dtd">
-->

<!ENTITY nbsp " ">

<!-- Parameter entities for use in ATTLIST declarations:
-->

<!ENTITY % lineSpecific "xml:space (default|preserve) ’preserve’" >

<!-- Information pool:

    %syntax;    Phrase-level elements that refer to Tcl syntactic entities.
    %inline;    All phrase-level elements
    %block;     Block-level elements
    %mixed;     All block- and phrase-level elements
    %structure; Structural elements that can appear directly in a MANPAGE

-->

<!ENTITY % syntax
    " META|M | LIT|L | I | B
    | BR
    | COMMAND | VARIABLE | METHOD | SYSCMD | FUNCTION | OPTION
    | WIDGET | SYSFUNC
    | OPT | GROUP
    ">

<!ENTITY % inline
    " #PCDATA
    | EMPH
    | REF
    | NEW
    | %syntax;
    ">

<!ENTITY % block
    " P | UL | OL | DL
    | EXAMPLE | SYNTAX | SYNOPSIS
    | METHODDEF
    | OPTIONDEF
    " >

<!ENTITY % mixed "%inline; | %block;">
```

```
<!ENTITY % structure
    " VARDEF
    | METHODLIST
    | OPTIONLIST
    | FUNCARG
    " >

<!-- ============================================================
        Top-level document structure:
     ============================================================ -->

<!ELEMENT TCLDOC        (SECTION*)                                      >
<!ELEMENT SECTION       (TITLE, ((MANPAGE|COMMENT)* | SECTION*))        >
<!ELEMENT TITLE         (#PCDATA)                                       >
<!ELEMENT MANPAGE
        (%mixed; | %structure; | NAME | DESC | HEADING | KEYWORDS | SEEALSO)*
>
<!-- %%% WANT:
    ELEMENT MANPAGE
        (NAME, DESC,
            SYNOPSIS?,
            (HEADING, (%structure;|%block;)*)*,
            KEYWORDS?,
            SEEALSO? )
-->
<!ELEMENT COMMENT       (#PCDATA)                                       >
<!-- %%% WANT:
        Change COMMENT element into a ?COPYRIGHT processing instruction.
-->

<!ATTLIST MANPAGE
    TYPE        CDATA   #REQUIRED
    EXTENSION   CDATA   #REQUIRED
>
<!ELEMENT NAME  (#PCDATA)       >
<!ELEMENT DESC  (%inline;)*     >

<!-- ============================================================
        Structural elements:
     ============================================================ -->

<!ELEMENT HEADING       (#PCDATA) >
<!ATTLIST HEADING
        ID      CDATA   #IMPLIED
>
<!-- Note: ID is declared CDATA rather than ID
     to allow more than one MANPAGE per document,
     since XML does not support SUBDOC.
-->

<!ELEMENT KEYWORDS      (#PCDATA) >
<!ATTLIST KEYWORDS
    NOTATION    CDATA   #FIXED "list of keywords (ws- or comma-separated)"
>
<!ELEMENT SEEALSO       (#PCDATA) >
<!ATTLIST SEEALSO
    NOTATION    CDATA   #FIXED "list of cross-references"
>
<!ELEMENT SYNOPSIS      (SYNTAX|NEW)+   >

<!-- ============================================================
        Block-level elements:
     ============================================================ -->

<!ELEMENT UL    (%mixed; | LI)* >
<!ELEMENT OL    (%mixed; | LI)* >
```

```
<!ELEMENT LI    EMPTY           >
<!-- %%% WANT:
        ELEMENT UL (LI+)
        ELEMENT OL (LI+)
        ELEMENT LI (%mixed;)*
-->

<!-- %%% DL content model is "(DT+, DD)*  +NEW", sort of -->
<!ELEMENT DL    ((DT, (DT|NEW)*, DD) | NEW)*    >
<!ELEMENT DT    (%inline;)*                     >
<!ELEMENT DD    (%mixed;)*                      >

<!ELEMENT P     EMPTY                           >
<!-- %%% WANT: ELEMENT P (%inline;)* -->

<!ELEMENT EXAMPLE       (%inline;)*                             >
<!ATTLIST EXAMPLE       %lineSpecific;                          >

<!ELEMENT SYNTAX        (#PCDATA | %syntax;)*                   >
<!ATTLIST SYNTAX        %lineSpecific;                          >

<!-- ============================================================
        Inline elements:
     ============================================================ -->

<!ELEMENT I     (%inline;)*     >       <!-- deprecate -->
<!ELEMENT B     (%inline;)*     >       <!-- deprecate -->
<!ELEMENT EMPH  (%inline;)*     >
<!ELEMENT BR    EMPTY   >
<!ELEMENT REF   (#PCDATA)       >
<!ATTLIST REF
    ID          CDATA           #REQUIRED
    PAGE        CDATA           #IMPLIED
    COMMAND     CDATA           #IMPLIED
    WIDGET      CDATA           #IMPLIED
    FUNCTION    CDATA           #IMPLIED
    NOTATION    CDATA           #FIXED "reference to HEADING"
>

<!-- ============================================================
        Syntax elements:
     ============================================================ -->

<!ELEMENT META  (#PCDATA|%syntax;)*     >       <!ELEMENT M (#PCDATA) >
<!ELEMENT LIT   (#PCDATA|%syntax;)*     >       <!ELEMENT L (#PCDATA) >
<!--  %%% WANT: ELEMENT (META|LIT) (#PCDATA) -->
<!ELEMENT FUNC          (#PCDATA)       >
<!ELEMENT INFO          (#PCDATA)       >
<!ELEMENT SYSCMD        (#PCDATA)       >
<!ELEMENT WIDGET        (#PCDATA)       >
<!ELEMENT OPT   (#PCDATA | %syntax;)*                   >
<!ELEMENT GROUP (#PCDATA | OR | %syntax;)*              >
    <!ELEMENT OR        EMPTY                                   >
<!ELEMENT COMMAND       (#PCDATA)       >
<!ATTLIST COMMAND
        PAGE    CDATA   #IMPLIED
>

<!ELEMENT FUNCTION      (#PCDATA)       >
<!ATTLIST FUNCTION
    PAGE        CDATA #IMPLIED
    NOTATION    CDATA #FIXED "Reference to MANPAGE TYPE=FUNC"
>

<!ELEMENT VARIABLE      (#PCDATA)       >
<!ATTLIST VARIABLE
```

```
        NAME    CDATA   #IMPLIED        >
<!-- %%% VARIABLE NAME is actually a hyperlink.  Only used once. -->

<!ELEMENT METHOD (#PCDATA)              >
<!ATTLIST METHOD
        NAME    CDATA   #IMPLIED
        COMMAND CDATA   #IMPLIED
        WIDGET  CDATA   #IMPLIED
        PAGE    CDATA   #IMPLIED
>
<!-- %%% METHOD attlist: documentation says "COMMAND/WIDGET/NAME";
     actually uses: PAGE/NAME.  PAGE is probably a bad idea.
-->

<!-- ============================================================
        Tcl-specific block-level elements:
     ============================================================ -->
<!ELEMENT VARDEF        (%mixed;)*      >
<!ATTLIST VARDEF
        NAME    CDATA   #REQUIRED       >

<!-- %%% WANT:
     ELEMENT METHODDEF  (SYNTAX, METHODDESC)
     ELEMENT METHODDESC (%mixed; | METHODDEF)*
-->
<!ELEMENT METHODDEF     (%mixed;)*      >
<!ATTLIST METHODDEF
        NAME    CDATA   #REQUIRED       >
<!ELEMENT METHODLIST    (#PCDATA)       >
<!ATTLIST METHODLIST
        NOTATION        CDATA #FIXED "List of methods"
>

<!ELEMENT OPTIONDEF     (%mixed;)*      >
<!ATTLIST OPTIONDEF
    SWITCH      CDATA   #REQUIRED
    ARG         CDATA   #IMPLIED
    NAME        CDATA   #IMPLIED
    WIDGET      CDATA   #IMPLIED
    CLASS       CDATA   #IMPLIED
    ALT         CDATA   #IMPLIED
>
<!ELEMENT OPTIONLIST    (#PCDATA)       >
<!ATTLIST OPTIONLIST
    SUPERCLASS  CDATA   #IMPLIED
    NOTATION    CDATA   #FIXED "List of options inherited from superclass"
>
<!ELEMENT OPTION        (#PCDATA)       >
<!ATTLIST OPTION
    COMMAND     CDATA   #IMPLIED
    WIDGET      CDATA   #IMPLIED
    SWITCH      CDATA   #IMPLIED
    PAGE        CDATA   #IMPLIED
    NOTATION    CDATA   #FIXED "reference to option"
>
<!--
    %%% OPTION: PAGE attribute not documented.
    %%% What is the SWITCH attribute?
-->

<!ELEMENT FUNCARG (ARGTYPE|ARGNAME|ARGDESC|NEW)*>
<!ATTLIST FUNCARG
    MODE        CDATA    #REQUIRED
>
<!--
    %%% MODE: (in | out | in/out)
```

```
    %%% Want: (in | out | inout)
-->
<!ELEMENT ARGTYPE       (#PCDATA)       >
<!ELEMENT ARGNAME       (#PCDATA)       >
<!ELEMENT ARGDESC       (%inline;)*     >

<!-- ============================================================
        Version control:
     ============================================================ -->
<!ELEMENT NEW   ANY >
<!ATTLIST NEW
    VERSION     CDATA   #IMPLIED
>

<!-- EOF -->
```

5 Testing Strategies

Our goal is to make the test suites for Tcl and extensions easy to create and maintain as well as invoke and interpret results. Uniformity across extension test suites can improve all of these things. For test suites to be uniform across extensions, they must have the same internal code structure and usage. They must also have the same (minimal) side effects. This outline contains our proposal for test suite specs that can help us achieve these goals.

5.1 Usage

5.1.1 Mechanism for Invocation

1. To run tests, use the following command line spec:
2. <shell> <testFile> ?<flag> ?<arg>?? ...
3. All test suites should share some basic command line options:

- All global variables that affect the test suite usage (such as VERBOSE, TESTS, and DO_ALL_TESTS should be accessible from the command line.
- Use the -tmpdir <dir> flag to specify the directory to put tmp files in. The default value is: [pwd]/testOutputDir
- Use the -match <globPatternList> flag to specify a value for the ::test::TESTS variable. The default value is {}.
- Use the -verbose <spec> flag to specify a value for the ::test::VERBOSE variable. VERBOSE can be set to any substring or permutation of "bps". The default value of VERBOSE is "b". If 'b' is present, then the entire test is printed for each failed test, otherwise only the test's name, desired output, and actual output, are printed for each failed test. If 'p' is present, then a line is printed for each passed test, otherwise no line is printed for passed tests. If 's' is present, then a line is printed for each skipped test, otherwise no line is printed for skipped tests.

5.1.2 Print Testing Statistics

Use the ::test::printStats command at the end of each test file to print the number of tests passed, failed, and skipped.

5.2 Structural Uniformity

5.2.1 Code structure

1. Global variables and procedures outside of the *.test files should be defined in the ::test namespace.
2. To make tests easy to skip, add appropriate flags (e.g., unixOnly) to any tests that should not always be run. For example, a test that should only be run on Unix should look like the following:

```
test getAttribute-1.1 {testing file permissions} {unixOnly} {
    lindex [file attributes foo.tcl] 5
```

```
        } {00644}
```

5.2.2 File Nomenclature and Structure

1. Regression tests (aka. glass-box or white-box tests) should reside in test files that reflect the file structure of the source code they are testing. They should match the glob pattern: <correspondingSourceFile>.test
2. Black-box tests should match the glob pattern: *_bb.test.
3. Where applicable, Tcl files that don't call the "test" command should match the pattern: *.tcl
4. Text files (with exception of README) are called: *.txt
5. Directory Heirarchy

```
                  tests
                    |
   --------------------------------------
   |     |       |                      |
README      all.tcl  defs.tcl  <same as source dir heirarchy>
                                        |
                              ---------------------
                              |             |     |
                           ?README?  all.tcl   *.test
```

5.2.3 Tests Should Not Rely on the Current Working Directory

1. Files that are part of the test suite should be accessed relative to ::test::testsDir (based on the location of defs.tcl).
2. Files that are created by the test suite should be accessed relative to ::test::tmpDir.

5.2.4 Tests Should Clean Up After Themselves

1. Test should not leave any new files or directories behind. If they must do so, the files should be left in the variable: ::test::tmpDir
2. Where possible, use the ::test::makeFile and ::test::makeDir commands to create files and directories.
3. Use the ::test::cleanupTests command at the end of each test file to remove files and directories created by ::test::makeFile and ::test::makeDir.

6 Distribution and Installation

Imagine an environment in which you had nothing but Tcl running and you typed *package require Expect*! Wouldn't it be cool if by some *automagical* stroke of genius, Tcl reached out over the Internet and grabbed the extension, downloaded it to your local machine, installed it into your Tcl installation, and proceeded to load and initialize Expect. Maybe—just maybe—if the binary for you particular platform was unavailable, the sources are downloaded and built for you. Carrying this idea further, Tcl extensions can be upgraded in very much the same way: *package upgrade* command? All the pain end users feel today of locating, downloading, building, and installing extensions, would all go away. Tcl Extension writers would then have a vehicle for getting their packages out to more users, more often in a smooth and simple manner.

A stepping stone for the above goal is to be able to give a Tcl extension install tool the URL of the location of where a packaged extension lives. The tool could interrogate a database at that URL to determine the best version of an extension against the version of Tcl you currently use. So if you were running Tcl 8.0.5, with the help of the database, the tool would determine it should retrieve TclX 8.0.5. The tool would download and install the extension into your Tcl instalaltion.

Possibly a shorter term goal than either of the above is to have the equivalent of .tar balls for Tcl extensions that the user can download, untar, and run a Tcl script that proceeds to install the particular extension.

A prerequisite to any of these automated installation strategies is having Tcl installed and working. For most platforms today, Tcl is installed from a source release. Tcl is likely to be installed using an OS specific installation format such as

Solaris *option package*, Linux *RPM*, and Windows *SETUP* application. [Should Scriptics have as a goal to provide binary distribution of Tcl on many platform in respective formats (very much like what CPAN has for Perl; CPAN sends you to a site that has a binary version of Perl for a particular platform; for example they send you to Sun to get the Solaris version of Perl)?]

A complete distribution and installation strategy includes two major area: the first, a packaging utility that puts together an extension including the complete manifest of files; the second, an installer that installs the extension into an end user's Tcl installation. The sections below discuss these two areas in further detail. It must be easy for a module developer to create both source and binary distributions.

Tcl and Tcl extensions should continue to support source distributions. It must be possible (and preferably easy) to as well download and build from a source distribution. (Necessary for users who don't happen to be using one of the major platforms, or who don't trust built distributions.) To aid in installing all modules, and in building extension from C/C++ source, there must be a standardized way to get Python's configuration data (such as compiler, compiler flags, platform, default library directory, etc.).

6.1 Packaging an Extension

There must be an easy, standardized way to create "built distributions" (ready-to-install downloadable resources, with all compilation and other processing done) for the major platforms.

Module distributions must have a standardized way to express their dependencies on other modules (both simple presence/absence and required version number) and on Tcl itself (version number), and these dependencies must be checked at install time to ensure that a proper version of Tcl is available.