library manual page - Tcl Built-In Commands

tcl-lang.org/man/tcl/TclCmd/library.htm

NAME

auto_execok, auto_import, auto_load, auto_mkindex, auto_qualify, auto_reset, tcl_findLibrary, parray, tcl_endOfWord, tcl_startOfNextWord, tcl_startOfPreviousWord, tcl_wordBreakAfter, tcl_wordBreakBefore — standard library of Tcl procedures

SYNOPSIS

auto_execok *cmd*
auto_import *pattern*
auto_load *cmd*
auto_mkindex *dir pattern pattern ...*
auto_qualify *command namespace*
auto_reset
tcl_findLibrary *basename version patch initScript enVarName varName*
parray *arrayName ?pattern?*
tcl_endOfWord *str start*
tcl_startOfNextWord *str start*
tcl_startOfPreviousWord *str start*
tcl_wordBreakAfter *str start*
tcl_wordBreakBefore *str start*

INTRODUCTION

Tcl includes a library of Tcl procedures for commonly-needed functions. The procedures defined in the Tcl library are generic ones suitable for use by many different applications. The location of the Tcl library is returned by the **info library** command. In addition to the Tcl library, each application will normally have its own library of support procedures as well; the location of this library is normally given by the value of the **$***app***_library** global variable, where *app* is the name of the application. For example, the location of the Tk library is kept in the variable **tk_library**.
To access the procedures in the Tcl library, an application should source the file **init.tcl** in the library, for example with the Tcl command

```
source [file join [info library] init.tcl]
```

If the library procedure **Tcl_Init** is invoked from an application's **Tcl_AppInit** procedure, this happens automatically. The code in **init.tcl** will define the **unknown** procedure and arrange for the other procedures to be loaded on-demand using the auto-load mechanism defined below.

COMMAND PROCEDURES

The following procedures are provided in the Tcl library:

auto_execok *cmd*

Determines whether there is an executable file or shell builtin by the name *cmd*. If so, it returns a list of arguments to be passed to **exec** to execute the executable file or shell builtin named by *cmd*. If not, it returns an empty string. This command examines the directories in the current search path (given by the PATH environment variable) in its search for an executable file named *cmd*. On Windows platforms, the search is expanded with the same directories and file extensions as used by **exec**. **Auto_execok** remembers information about previous searches in an array named **auto_execs**; this avoids the path search in future calls for the same *cmd*. The command **auto_reset** may be used to force **auto_execok** to forget its cached information.

auto_import *pattern*

Auto_import is invoked during **namespace import** to see if the imported commands specified by *pattern* reside in an autoloaded library. If so, the commands are loaded so that they will be available to the interpreter for creating the import links. If the commands do not reside in an autoloaded library, **auto_import** does nothing. The pattern matching is performed according to the matching rules of **namespace import**.

auto_load *cmd*

This command attempts to load the definition for a Tcl command named *cmd*. To do this, it searches an *auto-load path*, which is a list of one or more directories. The auto-load path is given by the global variable **auto_path** if it exists. If there is no **auto_path** variable, then the TCLLIBPATH environment variable is used, if it exists. Otherwise the auto-load path consists of just the Tcl library directory. Within each directory in the auto-load path there must be a file **tclIndex** that describes one or more commands defined in that directory and a script to evaluate to load each of the commands. The **tclIndex** file should be generated with the **auto_mkindex** command. If *cmd* is found in an index file, then the appropriate script is evaluated to create the command. The **auto_load** command returns 1 if *cmd* was successfully created. The command returns 0 if there was no index entry for *cmd* or if the script did not actually define *cmd* (e.g. because index information is out of date). If an error occurs while processing the script, then that error is returned. **Auto_load** only reads the index information once and saves it in the array **auto_index**; future calls to **auto_load** check for *cmd* in the array rather than re-reading the index files. The cached index information may be deleted with the command **auto_reset**. This will force the next **auto_load** command to reload the index database from disk.

auto_mkindex *dir pattern pattern ...*

Generates an index suitable for use by **auto_load**. The command searches *dir* for all files whose names match any of the *pattern* arguments (matching is done with the **glob** command), generates an index of all the Tcl command procedures defined in all the matching files, and stores the index information in a file named **tclIndex** in *dir*. If no pattern is given a pattern of ***.tcl** will be assumed. For example, the command

```
auto_mkindex foo *.tcl
```

will read all the **.tcl** files in subdirectory **foo** and generate a new index file **foo/tclIndex**.

Auto_mkindex parses the Tcl scripts by sourcing them into a child interpreter and monitoring the proc and namespace commands that are executed. Extensions can use the (undocumented) auto_mkindex_parser package to register other commands that can contribute to the auto_load index. You will have to read through auto.tcl to see how this works.

Auto_mkindex_old (which has the same syntax as **auto_mkindex**) parses the Tcl scripts in a relatively unsophisticated way: if any line contains the word "**proc**" as its first characters then it is assumed to be a procedure definition and the next word of the line is taken as the procedure's name. Procedure definitions that do not appear in this way (e.g. they have spaces before the **proc**) will not be indexed. If your script contains "dangerous" code, such as global initialization code or procedure names with special characters like **$**, *****, **[** or **]**, you are safer using **auto_mkindex_old**.

auto_reset
Destroys all the information cached by **auto_execok** and **auto_load**. This information will be re-read from disk the next time it is needed. **Auto_reset** also deletes any procedures listed in the auto-load index, so that fresh copies of them will be loaded the next time that they are used.

auto_qualify *command namespace*
Computes a list of fully qualified names for *command*. This list mirrors the path a standard Tcl interpreter follows for command lookups: first it looks for the command in the current namespace, and then in the global namespace. Accordingly, if *command* is relative and *namespace* is not **::**, the list returned has two elements: *command* scoped by *namespace*, as if it were a command in the *namespace* namespace; and *command* as if it were a command in the global namespace. Otherwise, if either *command* is absolute (it begins with **::**), or *namespace* is **::**, the list contains only *command* as if it were a command in the global namespace.
Auto_qualify is used by the auto-loading facilities in Tcl, both for producing auto-loading indexes such as *pkgIndex.tcl*, and for performing the actual auto-loading of functions at runtime.

tcl_findLibrary *basename version patch initScript enVarName varName*
This is a standard search procedure for use by extensions during their initialization. They call this procedure to look for their script library in several standard directories. The last component of the name of the library directory is normally *basenameversion* (e.g., tk8.0), but it might be "library" when in the build hierarchies. The *initScript* file will be sourced into the interpreter once it is found. The directory in which this file is found is stored into the global variable *varName*. If this variable is already defined (e.g., by C code during application initialization) then no searching is done. Otherwise the search looks in these directories: the directory named by the environment variable *enVarName*; relative to the Tcl library directory; relative to the executable file in the standard installation bin or bin/*arch* directory; relative to the executable file in the current build tree; relative to the executable file in a parallel build tree.

parray *arrayName ?pattern?*

Prints on standard output the names and values of all the elements in the array *arrayName*, or just the names that match *pattern* (using the matching rules of **string match**) and their values if *pattern* is given. *ArrayName* must be an array accessible to the caller of **parray**. It may be either local or global.

WORD BOUNDARY HELPERS

These procedures are mainly used internally by Tk.

tcl_endOfWord *str start*
Returns the index of the first end-of-word location that occurs after a starting index *start* in the string *str*. An end-of-word location is defined to be the first non-word character following the first word character after the starting point. Returns -1 if there are no more end-of-word locations after the starting point. See the description of **tcl_wordchars** and **tcl_nonwordchars** below for more details on how Tcl determines which characters are word characters.

tcl_startOfNextWord *str start*
Returns the index of the first start-of-word location that occurs after a starting index *start* in the string *str*. A start-of-word location is defined to be the first word character following a non-word character. Returns -1 if there are no more start-of-word locations after the starting point.

tcl_startOfPreviousWord *str start*
Returns the index of the first start-of-word location that occurs before a starting index *start* in the string *str*. Returns -1 if there are no more start-of-word locations before the starting point.

tcl_wordBreakAfter *str start*
Returns the index of the first word boundary after the starting index *start* in the string *str*. Returns -1 if there are no more boundaries after the starting point in the given string. The index returned refers to the second character of the pair that comprises a boundary.

tcl_wordBreakBefore *str start*
Returns the index of the first word boundary before the starting index *start* in the string *str*. Returns -1 if there are no more boundaries before the starting point in the given string. The index returned refers to the second character of the pair that comprises a boundary.

VARIABLES

The following global variables are defined or used by the procedures in the Tcl library. They fall into two broad classes, handling unknown commands and packages, and determining what are words.

AUTOLOADING AND PACKAGE MANAGEMENT VARIABLES

auto_execs
Used by **auto_execok** to record information about whether particular commands exist as executable files.

auto_index
Used by **auto_load** to save the index information read from disk.

auto_noexec
If set to any value, then **unknown** will not attempt to auto-exec any commands.

auto_noload
If set to any value, then **unknown** will not attempt to auto-load any commands.

auto_path
If set, then it must contain a valid Tcl list giving directories to search during auto-load operations (including for package index files when using the default **package unknown** handler). This variable is initialized during startup to contain, in order: the directories listed in the **TCLLIBPATH** environment variable, the directory named by the **tcl_library** global variable, the parent directory of **tcl_library**, the directories listed in the **tcl_pkgPath** variable. Additional locations to look for files and package indices should normally be added to this variable using **lappend**.

env(TCL_LIBRARY)
env(TCLLIBPATH)
If set, then it must contain a valid Tcl list giving directories to search during auto-load operations. Directories must be specified in Tcl format, using "/" as the path separator, regardless of platform. This variable is only used when initializing the **auto_path** variable.

WORD BOUNDARY DETERMINATION VARIABLES

These variables are only used in the **tcl_endOfWord**, **tcl_startOfNextWord**, **tcl_startOfPreviousWord**, **tcl_wordBreakAfter**, and **tcl_wordBreakBefore** commands.
tcl_nonwordchars
This variable contains a regular expression that is used by routines like **tcl_endOfWord** to identify whether a character is part of a word or not. If the pattern matches a character, the character is considered to be a non-word character. On Windows platforms, spaces, tabs, and newlines are considered non-word characters. Under Unix, everything but numbers, letters and underscores are considered non-word characters.

tcl_wordchars
This variable contains a regular expression that is used by routines like **tcl_endOfWord** to identify whether a character is part of a word or not. If the pattern matches a character, the character is considered to be a word character. On Windows platforms, words are comprised of any character that is not a space, tab, or newline. Under Unix, words are comprised of numbers, letters or underscores.

package manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/package.htm

NAME

package — Facilities for package loading and version control

SYNOPSIS

package forget ?*package package ...*?
package ifneeded *package version* ?*script*?
package names
package present *package* ?*requirement...*?
package present -exact *package version*
package provide *package* ?*version*?
package require *package* ?*requirement...*?
package require -exact *package version*
package unknown ?*command*?
package vcompare *version1 version2*
package versions *package*
package vsatisfies *version requirement...*
package prefer ?**latest**|**stable**?

DESCRIPTION

This command keeps a simple database of the packages available for use by the current interpreter and how to load them into the interpreter. It supports multiple versions of each package and arranges for the correct version of a package to be loaded based on what is needed by the application. This command also detects and reports version clashes.
Typically, only the **package require** and **package provide** commands are invoked in normal Tcl scripts; the other commands are used primarily by system scripts that maintain the package database.
The behavior of the **package** command is determined by its first argument. The following forms are permitted:

package forget ?*package package ...*?

Removes all information about each specified package from this interpreter, including information provided by both **package ifneeded** and **package provide**.

package ifneeded *package version* ?*script*?

This command typically appears only in system configuration scripts to set up the package database. It indicates that a particular version of a particular package is available if needed, and that the package can be added to the interpreter by executing *script*. The script is saved in a database for use by subsequent **package require** commands; typically, *script* sets up auto-loading for the commands in the package (or

calls **load** and/or **source** directly), then invokes **package provide** to indicate that the package is present. There may be information in the database for several different versions of a single package. If the database already contains information for *package* and *version*, the new *script* replaces the existing one. If the *script* argument is omitted, the current script for version *version* of package *package* is returned, or an empty string if no **package ifneeded** command has been invoked for this *package* and *version*.

package names
Returns a list of the names of all packages in the interpreter for which a version has been provided (via **package provide**) or for which a **package ifneeded** script is available. The order of elements in the list is arbitrary.

package present ?-exact? ***package ?requirement...?***
This command is equivalent to **package require** except that it does not try and load the package if it is not already loaded.

package provide ***package ?version?***
This command is invoked to indicate that version *version* of package *package* is now present in the interpreter. It is typically invoked once as part of an **ifneeded** script, and again by the package itself when it is finally loaded. An error occurs if a different version of *package* has been provided by a previous **package provide** command. If the *version* argument is omitted, then the command returns the version number that is currently provided, or an empty string if no **package provide** command has been invoked for *package* in this interpreter.

package require ***package ?requirement...?***
This command is typically invoked by Tcl code that wishes to use a particular version of a particular package. The arguments indicate which package is wanted, and the command ensures that a suitable version of the package is loaded into the interpreter. If the command succeeds, it returns the version number that is loaded; otherwise it generates an error.
A suitable version of the package is any version which satisfies at least one of the requirements, per the rules of **package vsatisfies**. If multiple versions are suitable the implementation with the highest version is chosen. This last part is additionally influenced by the selection mode set with **package prefer**.

In the “stable” selection mode the command will select the highest stable version satisfying the requirements, if any. If no stable version satisfies the requirements, the highest unstable version satisfying the requirements will be selected. In the “latest” selection mode the command will accept the highest version satisfying all the requirements, regardless of its stableness.

If a version of *package* has already been provided (by invoking the **package provide** command), then its version number must satisfy the *requirement*s and the command returns immediately. Otherwise, the command searches the database of information provided by previous **package ifneeded** commands to see if an acceptable version of the package is available. If so, the script for the highest acceptable version number is evaluated in the global namespace; it must do whatever is necessary to load the

package, including calling **package provide** for the package. If the **package ifneeded** database does not contain an acceptable version of the package and a **package unknown** command has been specified for the interpreter then that command is evaluated in the global namespace; when it completes, Tcl checks again to see if the package is now provided or if there is a **package ifneeded** script for it. If all of these steps fail to provide an acceptable version of the package, then the command returns an error.

package require -exact *package version*
This form of the command is used when only the given *version* of *package* is acceptable to the caller. This command is equivalent to **package require** *package version-version*.

package unknown ?*command*?
This command supplies a “last resort” command to invoke during **package require** if no suitable version of a package can be found in the **package ifneeded** database. If the *command* argument is supplied, it contains the first part of a command; when the command is invoked during a **package require** command, Tcl appends one or more additional arguments giving the desired package name and requirements. For example, if *command* is **foo bar** and later the command **package require test 2.4** is invoked, then Tcl will execute the command **foo bar test 2.4** to load the package. If no requirements are supplied to the **package require** command, then only the name will be added to invoked command. If the **package unknown** command is invoked without a *command* argument, then the current **package unknown** script is returned, or an empty string if there is none. If *command* is specified as an empty string, then the current **package unknown** script is removed, if there is one.

package vcompare *version1 version2*
Compares the two version numbers given by *version1* and *version2*. Returns -1 if *version1* is an earlier version than *version2*, 0 if they are equal, and 1 if *version1* is later than *version2*.

package versions *package*
Returns a list of all the version numbers of *package* for which information has been provided by **package ifneeded** commands.

package vsatisfies *version requirement...*
Returns 1 if the *version* satisfies at least one of the given requirements, and 0 otherwise. Each *requirement* is allowed to have any of the forms:

min
This form is called “min-bounded”.

min-
This form is called “min-unbound”.

min-max
This form is called “bounded”.

where “min” and “max” are valid version numbers. The legacy syntax is a special case of the extended syntax, keeping backward compatibility. Regarding satisfaction the rules are:

1. The *version* has to pass at least one of the listed *requirement*s to be satisfactory.
2. A version satisfies a “bounded” requirement when
    1. For *min* equal to the *max* if, and only if the *version* is equal to the *min*.
    2. Otherwise if, and only if the *version* is greater than or equal to the *min*, and less than the *max*, where both *min* and *max* have been padded internally with “a0”. Note that while the comparison to *min* is inclusive, the comparison to *max* is exclusive.
3. A “min-bounded” requirement is a “bounded” requirement in disguise, with the *max* part implicitly specified as the next higher major version number of the *min* part. A version satisfies it per the rules above.
4. A *version* satisfies a “min-unbound” requirement if, and only if it is greater than or equal to the *min*, where the *min* has been padded internally with “a0”. There is no constraint to a maximum.

package prefer ?latest|stable?
With no arguments, the commands returns either “latest” or “stable”, whichever describes the current mode of selection logic used by **package require**.
When passed the argument “latest”, it sets the selection logic mode to “latest”.

When passed the argument “stable”, if the mode is already “stable”, that value is kept. If the mode is already “latest”, then the attempt to set it back to “stable” is ineffective and the mode value remains “latest”.

When passed any other value as an argument, raise an invalid argument error.

When an interpreter is created, its initial selection mode value is set to “stable” unless the environment variable **TCL_PKG_PREFER_LATEST** is set. If that environment variable is defined (with any value) then the initial (and permanent) selection mode value is set to “latest”.

VERSION NUMBERS

Version numbers consist of one or more decimal numbers separated by dots, such as 2 or 1.162 or 3.1.13.1. The first number is called the major version number. Larger numbers correspond to later versions of a package, with leftmost numbers having greater significance. For example, version 2.1 is later than 1.3 and version 3.4.6 is later than 3.3.5. Missing fields are equivalent to zeroes: version 1.3 is the same as version 1.3.0 and 1.3.0.0, so it is earlier than 1.3.1 or 1.3.0.2. In addition, the letters “a” (alpha) and/or “b” (beta) may appear exactly once to replace a dot for separation. These letters semantically add a negative specifier into the version, where “a” is -2, and “b” is -1. Each may be specified only once, and “a” or “b” are mutually exclusive in a specifier. Thus 1.3a1 becomes (semantically) 1.3.-2.1, 1.3b1 is 1.3.-1.1. Negative numbers are not directly allowed in version specifiers. A version number not containing the letters “a” or “b”

as specified above is called a **stable** version, whereas presence of the letters causes the version to be called is **unstable**. A later version number is assumed to be upwards compatible with an earlier version number as long as both versions have the same major version number. For example, Tcl scripts written for version 2.3 of a package should work unchanged under versions 2.3.2, 2.4, and 2.5.1. Changes in the major version number signify incompatible changes: if code is written to use version 2.1 of a package, it is not guaranteed to work unmodified with either version 1.7.3 or version 3.1.

PACKAGE INDICES

The recommended way to use packages in Tcl is to invoke **package require** and **package provide** commands in scripts, and use the procedure **pkg_mkIndex** to create package index files. Once you have done this, packages will be loaded automatically in response to **package require** commands. See the documentation for **pkg_mkIndex** for details.

EXAMPLES

To state that a Tcl script requires the Tk and http packages, put this at the top of the script:

```
package require Tk
package require http
```

To test to see if the Snack package is available and load if it is (often useful for optional enhancements to programs where the loss of the functionality is not critical) do this:

```
if {[catch {package require Snack}]} {
    # Error thrown - package not found.
    # Set up a dummy interface to work around the absence
} else {
    # We have the package, configure the app to use it
}
```

pkg::create manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/packagens.htm

NAME

pkg::create — Construct an appropriate 'package ifneeded' command for a given package specification

SYNOPSIS

::pkg::create -name *packageName* **-version** *packageVersion* ?**-load** *filespec*? ... ?**-source** *filespec*? ...

DESCRIPTION

::pkg::create is a utility procedure that is part of the standard Tcl library. It is used to create an appropriate **package ifneeded** command for a given package specification. It can be used to construct a **pkgIndex.tcl** file for use with the **package** mechanism.

OPTIONS

The parameters supported are:

-name *packageName*
This parameter specifies the name of the package. It is required.

-version *packageVersion*
This parameter specifies the version of the package. It is required.

-load *filespec*
This parameter specifies a binary library that must be loaded with the **load** command. *filespec* is a list with two elements. The first element is the name of the file to load. The second, optional element is a list of commands supplied by loading that file. If the list of procedures is empty or omitted, **::pkg::create** will set up the library for direct loading (see **pkg_mkIndex**). Any number of **-load** parameters may be specified.

-source *filespec*
This parameter is similar to the **-load** parameter, except that it specifies a Tcl library that must be loaded with the **source** command. Any number of **-source** parameters may be specified.

At least one **-load** or **-source** parameter must be given.

pkg_mkIndex manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/pkgMkIndex.htm

NAME

pkg_mkIndex — Build an index for automatic loading of packages

SYNOPSIS

pkg_mkIndex ?*options...*? *dir* ?*pattern pattern ...*?

DESCRIPTION

Pkg_mkIndex is a utility procedure that is part of the standard Tcl library. It is used to create index files that allow packages to be loaded automatically when **package require** commands are executed. To use **pkg_mkIndex**, follow these steps:

1. Create the package(s). Each package may consist of one or more Tcl script files or binary files. Binary files must be suitable for loading with the **load** command with a single argument; for example, if the file is **test.so** it must be possible to load this file with the command **load test.so**. Each script file must contain a **package provide** command to declare the package and version number, and each binary file must contain a call to **Tcl_PkgProvide**.
2. Create the index by invoking **pkg_mkIndex**. The *dir* argument gives the name of a directory and each *pattern* argument is a **glob**-style pattern that selects script or binary files in *dir*. The default pattern is ***.tcl** and ***.[info sharedlibextension]**. **Pkg_mkIndex** will create a file **pkgIndex.tcl** in *dir* with package information about all the files given by the *pattern* arguments. It does this by loading each file into a child interpreter and seeing what packages and new commands appear (this is why it is essential to have **package provide** commands or **Tcl_PkgProvide** calls in the files, as described above). If you have a package split among scripts and binary files, or if you have dependencies among files, you may have to use the **-load** option or adjust the order in which **pkg_mkIndex** processes the files. See **COMPLEX CASES** below.

3. Install the package as a subdirectory of one of the directories given by the **tcl_pkgPath** variable. If **$tcl_pkgPath** contains more than one directory, machine-dependent packages (e.g., those that contain binary shared libraries) should normally be installed under the first directory and machine-independent packages (e.g., those that contain only Tcl scripts) should be installed under the second directory. The subdirectory should include the package's script and/or binary files as well as the **pkgIndex.tcl** file. As long as the package is installed as a subdirectory of a directory in **$tcl_pkgPath** it will automatically be found during **package require** commands.
   If you install the package anywhere else, then you must ensure that the directory containing the package is in the **auto_path** global variable or an immediate subdirectory of one of the directories in **auto_path**. **Auto_path** contains a list of directories that are searched by both the auto-loader and the package loader; by default it includes **$tcl_pkgPath**. The package loader also checks all of the subdirectories of the directories in **auto_path**. You can add a directory to **auto_path** explicitly in your application, or you can add the directory to your **TCLLIBPATH** environment variable: if this environment variable is present, Tcl initializes **auto_path** from it during application startup.

4. Once the above steps have been taken, all you need to do to use a package is to invoke **package require**. For example, if versions 2.1, 2.3, and 3.1 of package **Test** have been indexed by **pkg_mkIndex**, the command **package require Test** will make version 3.1 available and the command **package require -exact Test 2.1** will make version 2.1 available. There may be many versions of a package in the various index files in **auto_path**, but only one will actually be loaded in a given interpreter, based on the first call to **package require**. Different versions of a package may be loaded in different interpreters.

OPTIONS

The optional switches are:

-direct

The generated index will implement direct loading of the package upon **package require**. This is the default.

-lazy

The generated index will manage to delay loading the package until the use of one of the commands provided by the package, instead of loading it immediately upon **package require**. This is not compatible with the use of *auto_reset*, and therefore its use is discouraged.

-load *pkgPat*

The index process will preload any packages that exist in the current interpreter and match *pkgPat* into the child interpreter used to generate the index. The pattern match uses string match rules, but without making case distinctions. See **COMPLEX CASES** below.

-verbose
Generate output during the indexing process. Output is via the **tclLog** procedure, which by default prints to stderr.

--
End of the flags, in case *dir* begins with a dash.

PACKAGES AND THE AUTO-LOADER

The package management facilities overlap somewhat with the auto-loader, in that both arrange for files to be loaded on-demand. However, package management is a higher-level mechanism that uses the auto-loader for the last step in the loading process. It is generally better to index a package with **pkg_mkIndex** rather than **auto_mkindex** because the package mechanism provides version control: several versions of a package can be made available in the index files, with different applications using different versions based on **package require** commands. In contrast, **auto_mkindex** does not understand versions so it can only handle a single version of each package. It is probably not a good idea to index a given package with both **pkg_mkIndex** and **auto_mkindex**. If you use **pkg_mkIndex** to index a package, its commands cannot be invoked until **package require** has been used to select a version; in contrast, packages indexed with **auto_mkindex** can be used immediately since there is no version control.

HOW IT WORKS

Pkg_mkIndex depends on the **package unknown** command, the **package ifneeded** command, and the auto-loader. The first time a **package require** command is invoked, the **package unknown** script is invoked. This is set by Tcl initialization to a script that evaluates all of the **pkgIndex.tcl** files in the **auto_path**. The **pkgIndex.tcl** files contain **package ifneeded** commands for each version of each available package; these commands invoke **package provide** commands to announce the availability of the package, and they setup auto-loader information to load the files of the package. If the **-lazy** flag was provided when the **pkgIndex.tcl** was generated, a given file of a given version of a given package is not actually loaded until the first time one of its commands is invoked. Thus, after invoking **package require** you may not see the package's commands in the interpreter, but you will be able to invoke the commands and they will be auto-loaded.

DIRECT LOADING

Some packages, for instance packages which use namespaces and export commands or those which require special initialization, might select that their package files be loaded immediately upon **package require** instead of delaying the actual loading to the first use of one of the package's command. This is the default mode when generating the package index. It can be overridden by specifying the **-lazy** argument.

COMPLEX CASES

Most complex cases of dependencies among scripts and binary files, and packages being split among scripts and binary files are handled OK. However, you may have to adjust the order in which files are processed by **pkg_mkIndex**. These issues are described in detail below.
If each script or file contains one package, and packages are only contained in one file, then things are easy. You simply specify all files to be indexed in any order with some glob patterns.

In general, it is OK for scripts to have dependencies on other packages. If scripts contain **package require** commands, these are stubbed out in the interpreter used to process the scripts, so these do not cause problems. If scripts call into other packages in global code, these calls are handled by a stub **unknown** command. However, if scripts make variable references to other package's variables in global code, these will cause errors. That is also bad coding style.

If binary files have dependencies on other packages, things can become tricky because it is not possible to stub out C-level APIs such as **Tcl_PkgRequire** API when loading a binary file. For example, suppose the BLT package requires Tk, and expresses this with a call to **Tcl_PkgRequire** in its **Blt_Init** routine. To support this, you must run **pkg_mkIndex** in an interpreter that has Tk loaded. You can achieve this with the **-load** *pkgPat* option. If you specify this option, **pkg_mkIndex** will load any packages listed by **info loaded** and that match *pkgPat* into the interpreter used to process files. In most cases this will satisfy the **Tcl_PkgRequire** calls made by binary files.

If you are indexing two binary files and one depends on the other, you should specify the one that has dependencies last. This way the one without dependencies will get loaded and indexed, and then the package it provides will be available when the second file is processed. You may also need to load the first package into the temporary interpreter used to create the index by using the **-load** flag; it will not hurt to specify package patterns that are not yet loaded.

If you have a package that is split across scripts and a binary file, then you should avoid the **-load** flag. The problem is that if you load a package before computing the index it masks any other files that provide part of the same package. If you must use **-load**, then you must specify the scripts first; otherwise the package loaded from the binary file may mask the package defined by the scripts.

package index script interface guidelines

In a recent TCLCORE message, there was a request to document what the interface guarantees are between a package index script and a package unknown callback. This page created to record such documentation.

LV what is meant by "package index script" - code such as pkgIndex.tcl?

DGP The default package unknown callback, [tcl_PkgUnknown] finds the index scripts for the packages it manages in files named **pkgIndex.tcl**, so yes, that's the most common example of an index script.

Other package unknown callbacks might choose to retrieve or generate their index scripts from other sources, or by other methods.

Probably most important is that a package index script should **never** raise an error.

This can be tricky because index scripts can be evaluated in just about any kind of Tcl interpreter, by any registered [package unknown] manager, so you should not depend on things you might depend on in your more day-to-day Tcl programming.

Especially noteworthy along those lines is that a package index script might be evaluated in a Tcl interpreter for any release of Tcl from 7.5 on. This means you should not be using any Tcl 8 features in your index script until after the index script itself verifies that the interp has a recent enough Tcl in it.

Index scripts can rely on the [return] command to cleanly terminate evaluation of the index script. So, a useful technique in an index script is to check the interp for a recent enough Tcl release to support the rest of the index script, and to support the package indexed by the index script:

```
if {![package vsatisfies [package provide Tcl] 8]} {return}
```

Likewise, an index script should not raise any of Tcl's other return codes like [break] or [continue]. It's best to think of those as causing undefined behavior, and just avoid them completely in index scripts. The actual behavior will probably depend on the internal details of the [package unknown] handler that the index script author really shouldn't know about, let alone depend on.

An index script can also depend on all of the commands built in to Tcl being available by their simple names. This does **not** mean the index script is evaluated in the global namespace. It only means that non-buggy [package unknown] handlers will not mask Tcl's built-in commands in the context they provide for index script evaluation.

An index script can rely on the existence of a variable known in the current context as *dir*. The contents of that variable are the absolute file system path to the installation directory associated with the package indexed by this index script. It is up to the [package unknown] manager to perform this initialization. Thus the [package unknown] manager keeps track of what installation directory goes with what package goes with what index script. The default [package unknown] handler, [tclPkgUnkown] achieves this by assigning to *dir* the name of the directory that contains a file named *pkgIndex.tcl* that contains the index script.

Since the index script does not know what namespace context or proc context it might be evaluated in, if it wants to access a global variable, it should either use the [global] command to bring that variable into current scope, or use a fully-qualified name for the variable (*::tcl_platform*) after verifying a Tcl 8 interpreter.

An index script should **not** call [package require] for any package. Evaluation of an index script is not about loading any package, it is only about registering a load script for later use. Other than performing that registration by calling [package ifneeded], an index script should strive to be free of side-effects.

An index script should not assume it is kept in a file and is evaluated by [source]. This means the index script should not depend on [info script] to return anything useful. The *dir* variable is the interface to use to discover the installation directory of the package, and that's all the index script should need to know.

Don't do this:

```
 package ifneeded foo 1.0 "load [file join $dir foo[info sharedlibextension]]"
```

That will break if *$dir* contains spaces. Do this instead:

```
 package ifneeded foo 1.0 [list load [file join $dir foo[info
sharedlibextension]]]
```

Be sure to read the discussion at pkgIndex.tcl which provides a Tcl 8.5 alternative preferred by DKF.

On 5-jan-2004 in c.l.t., Michael Schlenker explains with respect to:

`Probably most important is that a package index script should **never** raise an error.'

The problem is mainly with errors thrown when package require looks for packages, not when they are found and are about to be loaded.

i.e. the following example code is deemed OK:

```
proc loadMyPackage {dir} {
   if { ![CheckRequirement] } {

       # this is OK
       return -code error $errMsg

   }
   source [file join $dir sourceFile.tcl]
}
package ifneeded myPackage [list loadMyPackage $dir]
```

Of course, the error could also be raised inside the sourceFile.tcl.

DGP The problem with either of those approaches is they do not play well with package's handling of multiple installed versions of a package. Essentially, the [package ifneeded] tells the [package] system that a package is available to be loaded, even though the attempt to load it is going to produce an error. If this index script is for myPackage version 1.2, [package] is going to prefer loading it over myPackage version 1.1, which might not have the "error on load" problem. That would mean we're ignoring a package that works in favor of one that doesn't. It's best if you can avoid [package ifneeded] registration of packages that you know cannot successfully load in the current interp.

In an environment where you can be sure there's only one version of your package "installed" (the internals of a Starkit, perhaps?), you can probably get away with that. But in that environment a more generally correct index script will also work, so why not?

Another, perhaps less serious, problem is that the example does have the side effect of creating a [loadMyPackage] command.

Lars H: If the need is to perform several commands for loading a package, then the helper proc is quite unnecessary. A multiline package ifneeded script is straightforward to construct using format:

```
 package ifneeded myPackage 1.0 [format {
     package require sourceWithEncoding
     sourceWithEncoding utf-8 [file join %s myPackage.tcl]
 } [list $dir]]
```

Note how the helper **sourceWithEncoding** package is not loaded until **myPackage** is actually required. Also note that dir is list-quoted before it is passed to format, since it will appear as a complete word in the script (this is similar to how bind percent substitution works).

NEM: Or a lambda (in Tcl 8.5+):

```
if {![package vsatisfies [package provide Tcl] 8.5]} {return}
package ifneeded myPackage 1.0 [list apply {dir {
   package require sourceWithEncoding
   sourceWIthEncoding utf-8 [file join $dir myPackage.tcl]
}} $dir]
```

(Of course, for this specific example there is now [source -encoding]).

Lars H: The big disadvantage of that is of course that it's only suitable for packages that require Tcl 8.5. An advantage is that it lets you create variables that are local to the script.

TIP 55: Package Format for Tcl Extensions

core.tcl-lang.org/tips/doc/trunk/tip/55.md

Abstract

This document specifies the contents of a binary distribution of a Tcl package, especially directory structure and required files, suitable for automated installation into an existing Tcl installation.

Rationale

There is currently no standard way of distributing or installing a Tcl extension package. The TEA document defines a standard interface to *building* packages and includes an *install* target but presumes that the packages is being installed on the same machine as it was built. This TIP defines a directory structure and assorted files for the binary distribution of a package which can be placed into an archive (for example zip or tar file) and transferred for installation on another machine. A basic mechanism for installation of packages is also described.

Definitions

The following definitions are excerpted from [78]:

package: A collection of files providing additional functionality to a user of a Tcl interpreter when loaded into said interpreter.

> Some files in a package implement the provided functionality whereas other files contain metadata required by the package management of Tcl to be able to use the package.

distribution: An encapsulation of one or more *packages* for transport between places, machines, organizations, and people.

shared library: A piece of binary code that provides a set of operations and data structures like a normal library, but which does not need to be physically incorporated into the executables that use it until they are actually executed. This is the normal way to distribute binary code for a Tcl package such that it can be incorporated into a Tcl interpreter with the *load* command. On Windows, shared libraries are known as DLLs, on the Macintosh ...

References

Much of the required structure for an installable distribution is defined by the requirements of Tcl's existing package loading methods. The structure of an installable distribution should largely mirror the structure of an installed package where possible.

The R system (a statistical package http://www.r-project.org/) has a well defined package format which enables automatic installation of new packages and integration of documentation and demonstration programs for these with that of the main R system.

A number of packaging and installation systems (for example, Debian http://www.debian.org and RPM http://www.redhat.com) have been developed by the Linux community which provide an interesting range of facilities. These systems commonly provide facilities for pre and post installation scripts and pre and post removal scripts to help set up and shut down packages. Also included are detailed dependency relations between packages which can be used by an installer to ensure that a package will work once it is installed or warn of potential conflicts after installation.

A significant part of this proposal is the proposed format of the package metadata which derives from other metadata standardisation efforts, mainly the Dublin Core http://purl.org/dc/ and the Resource Description Framework http://www.w3.org/RDF .

Requirements

The simplest case of a Tcl package is one that contains only Tcl code; these will be considered first, and the additional issues raised by packages containing compiled code will be dealt with later.

The minimum contents of a Tcl only package are defined by the requirements of [package require xyzzy]. The package needs to be placed in a directory on the *auto_path* and must contain one or more *.tcl* files which implement the functionality provided by the package.

In addition to these files, it is useful to include documentation for the commands implemented by the package and some additional metadata about the author etc. Distributions might also optionally include demonstration scripts and applications illustrating their use, these could either be incorporated into the documentation or included as stand-alone Tcl files.

Distributions which include shared libraries add an additional layer of complexity since these will only run on the platforms for which they have been compiled. There are two clear options here: either distributions are platform specific, intended for installation on one platform alone, or the structure of the distribution is extended to allow the option of including multiple shared libraries. The latter option would allow a single installation to serve multiple platforms and so should be preferred although this TIP will not *require* a distribution to support multiple platforms.

Proposed Directory Structure

The following directory structure is proposed for an installable distribution:

```
packagename$version
    + DESCRIPTION.txt  -- Metadata, description of the package
    + doc/             -- documentation
    + examples/        -- example scripts and applications
    + $architecture/   -- shared library directories
    + pkgIndex.tcl     -- package index file (optional)
```

In addition, a distribution may include any additional files or directories required for its operation.

DESCRIPTION is a file containing metadata about the package(s) contained in the distribution. Its format will be described in a later section of this document.

The file *pkgIndex.tcl* currently required by the package-loading mechanism of the Tcl core is *optionally* distributed. In most cases, it will be generated by the installer; all the information which is necessary to do this is part of the distribution. Distribution authors should only include *pkgIndex.tcl* if special features of their distribution mean that the generated file would not work.

If the *pkgIndex.tcl* file is included in the distribution it should load files from their locations within the distribution directory structure. For example, Tcl files should be loaded from the *tcl* directory.

doc/ directory contains documentation in an accepted format. Currently Tcl documentation is delivered either in source form (nroff or TMML) or as HTML files. Given the lack of a standard cross platform solution, this TIP does not require a specific format; however, the inclusion of either a text or HTML formatted help file is strongly encouraged. If HTML formatted help is included the main file should be named *index.html* or *index.htm* so that it can be linked to a central web page. If only plain text documentation is included there should be a file called *readme.txt* (in either upper or lower case) which will serve as the top level documentation file.

examples/ directory contains one or more Tcl files giving examples of the use of this package. These should be complete scripts suitable for either sourcing in tclsh/wish or running from the command line. The examples should be self contained and any external data should be included in files in this directory or a sub-directory. This directory should contain a file *readme.txt* which explains how to run the examples and provides a commentary on what they do.

$architecture directories contain shared libraries for various platforms. The special architecture *tcl* is used for Tcl script files. They either implement the package or contain companion procedure definitions to the shared libraries of the package.

The distribution need not provide all possible combinations of architectures and may only provide one shared library. This structure is proposed to allow shared libraries to co-exist in a multi-platform environment and to allow binary packages to be distributed in multi-platform distributions. The architectures included in the distribution should be named in the DESCRIPTION.txt file.

The possible values of $architecture and methods for generating them are discussed in a later section.

Metadata

This section defines the metadata describing the package contained in the distribution in a format-neutral way. The model for this data is that provided by the Resource Description Framework (RDF http://www.w3.org/rdf) which defines a triple based data model. The RDF model defines objects, their properties and relationships between them. In addition, where possible, element names are taken from the Dublin Core Metadata Element Set http://dublincore.org/documents/1999/07/02/dces/ which defines a standard set of element names for metadata. Dublin Core names are marked with DC in parentheses in the following list.

In a package description, the object being described is the package itself, hence the element names are all intended to describe packages. Other objects might be described including people and organisations. The package description should not include these objects but a package repository might store them separately keyed on the values stored in this description (e.g. email addresses of creators).

- *Identifier* (DC)

  > This element is a string containing the name of the distributed package. The name may consist only of alphanumeric characters, colons, dashes and underscores. This name should correspond to the name of the package defined by this distribution (that is, the code should contain *package provide xyzzy* where *xyzzy* is the value of this element.
  >
  > Care must be taken to make this name unique among the package names in the archive. To overcome this, namespace style names separated by double colons should be used.
  >
  > Examples: xyzzy, tcllib, xml::soap, cassidy::wonderful-package_2

- *Version*

  > This element is a string containing the version of the package. It consists of 4 components separated by full stops. The components are *major version*, *minor version*, *maturity* and *level*; and are written in this order.
  >
  > The major and minor version components are integer numbers greater than or equal to zero.
  >
  > The component *maturity* is restricted to the values a, b. The represent the maturity states *alpha*, *beta* respectively. For a production release, this component can be omitted.
  >
  > The *level* component allows a more fine-grained differentiation of maturity levels. When a package has maturity *production* the *level* component is often called the *patchlevel* of the package. If the *level* component is zero, it may be omitted.
  >
  > The period each side of the *maturity* component may be omitted.
  >
  > Valid version numbers can be decoded via the following regular expression:

```
regexp {([0-9]+)\.([0-9]+)\.?([ab])?\.?([0-9]*)} $ver => major minor maturity
level
```

  > Examples: 8.4.0 8.4a1 2.5.b.5

- *Title* (DC)

  > This element is a free form string containing a one sentence description of the package contained in the distribution.
  >
  > Example: Installer Tools for Tcl Packages

- *Creator* (DC)

  > This element is a string containing the name of the person, organisation or service responsible for the creation of the package optionally followed by the email address of the author in angle brackets http://www.faqs.org/rfcs/rfc2822.html . More detail about an author can be provided in a separate object in the RDF description and if this is provided the email address should be used as the value of the Name field in that object.
  >
  > If there is more than one author this field may appear multiple times.
  >
  > Email addresses may be obfuscated to avoid spam harvesters.
  >
  > Example: Steve Cassidy

- *Contributor* (DC)

  > This element is a string analogous to the Creator element which contains the name of a contributor to the package.

- *Rights* (DC)

  > Typically, a Rights element will contain a rights management statement for the resource, or reference a service providing such information. This will usually be a reference to the license under which the package is distributed. This can be a free form string naming the license or a URL referring to a document containing the text of the license.
  >
  > If the Rights element is absent, no assumptions can be made about the status of these and other rights with respect to the resource.
  >
  > Examples: BSD, http://www.opensource.org/licenses/artistic-license.html

- *URL*

  > This element is a string containing an url referring to a document or site at which the information about the package can be found. This url is *not* the location of the distribution, as this might be part of a larger repository separate from the package site.
  >
  > Example: http://www.shlrc.mq.edu.au/~steve/tcl/

- *Available* (DC)

  > This element is the release data of the package in the form YYYY-MM-DD.
  >
  > YYYY is a four-digit integer number greater than zero denoting the year the distribution was released.
  >
  > MM is a two-digit integer number greater than zero and less than
  >
  > > It is padded with zero at the front if it less than 10. It denotes the month the distribution was released. The number 1 represents January, 2 represents February; and 12 represents December.
  >
  > DD is a two-digit integer number greater than zero and less than 32. It is and padded with zero at the front if less than 10. It denotes the day in the month the distribution was released.
  >
  > A valid data string can be obtained with the Tcl command [clock format [clock seconds] -format "%Y-%m-%d"]
  >
  > Example: 2002-01-23
  >
  > (The DC element is Date but it can be refined to Created, Available, Applies)

- *Description* (DC)

  > This element is a free form string briefly describing the package.

- *Architecture*

  > This element is a string describing one of the architectures included in the distribution. As a distribution is allowed to contain the files for several architectures, this element may appear multiple times and should correspond to a directory in the distribution.

- *Require*

  > Names a package that must be installed for this package to operate properly. This should have the same format as the *package require* command, eg. *?-exact? package ?version?*.
  >
  > Example: http 2.0

- *Recommend*

  > Declares a strong, but not absolute dependency on another package. In most cases this package should be installed unless the user has specific reasons not to install them.

- *Suggest*

  > Declares a package which would enhance the functionality of this package but which is not a requirement for the basic functionality of the package.

- *Conflict*

  > Names a package with which can't be installed alongside this package. The syntax is the same as for Require. If a conflicting package is present on the system, an installer might offer an option of removing it or not installing this package.

- *Subject* (DC)

  > The topic or content of the package expressed as a set of Keywords. At some future time, a set of canonical keywords may be established by a repository manager.

The following Dublin Core elements were not included in the standard set above but may be used in a package description if appropriate.

- *Publisher*

  > An entity responsible for making the package available.

- *Type*

  > The nature or genre of the content of the resource. For a Tcl package the value of this element would be Software if the DCMI Type Vocabulary http://au.dublincore.org/documents/2000/07/11/dcmi-type-vocabulary/ was used. A more useful set of types might be developed in the future for Tcl packages.

- *Format*

  > The physical or digital manifestation of the resource. This might be used by archive maintainers to specify the format of a package archive, eg. zip, tar etc.

- *Source*

  > A Reference to a resource from which the present resource is derived.

- *Language*

  > A language of the intellectual content of the resource. Could be used if multi-language packages are available. Should use the two letter language code defined by RFC 1766, eg. 'fr' for French, 'en' for English.

Encoding of the Metadata

The primary means of storing RDF data is using XML but it can be stored in many other formats. This TIP prescribes a simple text based encoding according to the RFC 2822 format which is described in this section. Data stored in this format can be converted to XML format for use by other tools, similarly XML formatted descriptions can be converted into this text format without loss of information.

The text format description is stored in the file *DESCRIPTION.txt*. The XML formatted version of the data may be stored in the file *DESCRIPTION.rdf* within the archive and may be automatically generated if not present.

The general format of this file is that of a RFC 2822 mail message, without body and using custom headers. The available headers are the case-independent logical names from the preceding section but may be augmented by other fields defined by repository maintainers or other applications. The headers are allowed appear in any order.

Example:

```
Identifier: stemmer
Version: 1.0.0
Title: A stemmer for English.
Creator: Steve Cassidy <[email protected]>
Description:   Provides a procedure to remove any prefixes or suffixes on
       a word to give the word stem. Uses Porter's algorithm to do this
       in an intelligent manner with an accuracy of around 80%.
Rights: BSD
URL: http://www.shlrc.mq.edu.au/emu/tcl/
Available: 2001-08-16
Architecture: tcl
Subject: linguistics
Subject: text
```

Combination Distributions

It is often useful to combine a number of related packages so that they can be installed together to provide a certain kind of functionality, for example, web page production tools or database access. Perl uses the term *Bundle* to refer to such a group of related packages. There are two alternative mechanisms for distribution of such a package within the mechanisms suggested here. Firstly, since a distribution may contain more than one package, the set of files making up the various packages could be combined together and described by a single DESCRIPTION.txt file. This is similar to the way that tcllib is currently distributed. The disadvantage would be that all of the Tcl files implementing these packages would have to reside in the same directory which could cause name clashes.

The second alternative is to create a distribution consisting of only a DESCRIPTION.txt file to describe which Requires the component packages causing them to be installed from the repository. For example, tcllib might be described as follows:

```
Identifier: tcllib
Version: 1.0.0
Title: The Standard Tcl Library
Description:  This package is intended to be a collection of Tcl
           packages that provide utility functions useful to a large
           collection of Tcl programmers.
Rights: BSD
URL: http://sourceforge.net/projects/tcllib
Contributor: Andreas Kupries  <andreas_kupries at users dot sourceforge dot net>
Contributor: Don Porter <dgp at users dot sourceforge dot net>
Require: base64
Require: cmdline
Require: csv
...
```

Installing tcllib would cause the installer to fetch base64, cmdline, csv etc from the repository and install them in order to satisfy the tcllib requirement. A new pkgIndex.tcl file could be constructed to load all of these packages if *[package require tcllib]* was called.

Architecture

Possible values for $architecture in the directory structure include:

- the value of *tcl_platform(platform)*: windows, unix, macintosh
- a composite of tcl_platform values: *$tcl_platform(machine)-$tcl_platform(os)-$tcl_platform(osVersion)*
- a canonical system name as returned by *config.guess*: *i686-pc-linux-gnu*

Installing Packages

A package structured according to this TIP can be installed using the following steps:

1. Download the package archive (eg. zip file)
2. Locate a writable directory included on $auto_path (or ask for a installation directory)
3. Unpack the archive in the desired location.
4. Run pkg_mkIndex with appropriate arguments to generate a pkgIndex.tcl file if none is present. Arguments will include the appropriate Architecture directories for the platform.
5. *(optional)* link help files and demos to the central index.

Alternatives

Alternatives might be considered for the package DESCRIPTION.txt file, for the documentation directory and for the location of shared libraries.

An alternative for package description file is to include an alternative package description, for example the XML based ``ppd_ format used to describe Perl packages on the ActiveState Perl package repository. The main motivation for the simple format proposed is that it is trivial for authors to write and trivial for programs to read and can be transformed into standards based RDF XML. The use of the DC element names means that search engines etc. will be able to usefully index the packages in a repository.

Note that the ppd format could still be used to describe packages stored in a repository for installation and that some of the information required to build the ppd format could be derived from the description file.

In the R package format referenced earlier, documentation is included in a standard source form and is converted to HTML or text based help pages; these might be included in the package or derived from the source forms on installation. The closest option for Tcl would be to require nroff format help files which can be converted to HTML or text files on installation. Unfortunately there is no guaranteed tool to do nroff->X conversion on Windows or Macintosh platforms. Until there is an accepted way of authoring Tcl documentation this TIP defers any standard layout of these files in an installable package.

The alternative to having shared libraries in specific directories is to have separate packages for each new platform. This has the advantage of making the packages smaller and more closely correspond to the existing directory structure of an installed package. The main motivation for the suggested directory structure is to allow multi-platform packages or to facilitate multi-platform installations.

Supporting Tools

The standards outlined in this TIP should be supported by Tcl scripts to:

- Generate empty package templates for new projects.
- Validate package directories or archive files.
- Read and write the DESCRIPTION.txt file and provide a standard interface to the information it contains. Convert between RFC 2822 and XML formats.
- Install a package from an appropriately structured archive.

In addition, the TEA standard should be extended with a *package* makefile target which will act like the current *install* target but which will copy files to a local directory and optionally build an archive of the package for distribution.