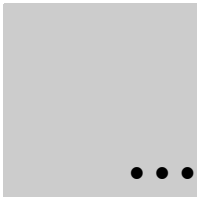


TclPro™ User's Guide

Scriptics Corporation
Version 1.3

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

http://www.scriptics.com

ACKNOWLEDGEMENTS
Michael McLennan is the primary developer of [incr Tcl] and [incr Tk]. Jim Ingham and Lee Bernhard handled the Macintosh and Windows ports of [incr Tcl] and [incr Tk]. Mark Ulferts is the primary developer of [incr Widgets], with other contributions from Sue Yockey, John Sigler, Bill Scott, Alfredo Jahn, Bret Schuhmacher, Tako Schotanus, and Kris Raney. Mark Diekhans and Karl Lehenbauer are the primary developers of Extended Tcl (TclX). Don Libes is the primary developer of Expect.

TclPro Wrapper incorporates compression code from the Info-ZIP group. There are no extra charges or costs in TclPro due to the use of this code, and the original compression sources are freely available from *http://www.cdrom.com/pub/infozip* or *ftp://ftp.cdrom.com/pub/infozip*.

NOTE: TclPro is packaged on this CD using Info-ZIP's compression utility. The installation program uses UnZip to read zip files from the CD. Info-ZIP's software (Zip, UnZip, and related utilities) is free and can be obtained as source code or executables from Internet WWW sites including *http://www.cdrom.com/pub/infozip*.

Contents

Chapter 1
Introduction to TclPro

The TclPro™ development environment is a set of powerful development tools and extended Tcl platform for professional Tcl developers. TclPro 1.3 consists of:

TclPro Debugger
: Find bugs quickly with features including breakpoints, single-stepping, stack and variable display, and variable-based breakpoints.

TclPro Checker
: Scan your Tcl scripts to identify a variety of potential problems including syntax errors, misuses of the Tcl and Tk built-in commands, and potential performance and portability problems. TclPro Checker also helps you to upgrade from older versions of Tcl to the latest releases by locating potential compatibility problems.

TclPro Compiler
: Translate your Tcl scripts into bytecode files so that you can distribute your applications without providing access to the original Tcl source code. TclPro Compiler protects your intellectual property and prevents customers from modifying your scripts.

TclPro Wrapper
: Create a single executable file containing everything needed to run a Tcl application. TclPro Wrapper makes it easy to distribute Tcl applications to your users and manage upgrades in Tcl versions.

Tcl/Tk 8.2
: The latest version of Tcl/Tk is pre-compiled and ready for use.

Bundled extensions
: Several popular Tcl extensions—[incr Tcl], [incr Tk], TclX, and Expect—are pre-compiled for all of the TclPro supported platforms. The TclPro tools have built-in support for all bundled extensions.

Enhanced interpreters

The **protclsh** and **prowish** Tcl interpreters include built-in support for all bundled extensions and the Tcl bytecode files produced by TclPro Compiler.

TclPro supports the following platforms:

- Windows
- Solaris (SPARC)
- HP-UX
- Irix
- Linux (Intel/glibc2)

See the online release notes for specific operating system versions supported.

Installing TclPro

This section describes how to install TclPro from either a CD or the Scriptics Web site, install the Adobe Acrobat Reader (if needed), and enter your license information so you can run the TclPro applications.

Installing TclPro from CD

The TclPro CD contains installer applications that make installing TclPro very easy. All you need to do is run the *setup.sh* (or *setup.exe* on Windows) program located at the top level of the CD-ROM. The installation program will guide you through the various installation options. Please see the *TclPro Installation Guide* that ships with this release for platform-specific details and additional instructions about installing from the CD-ROM.

Installing TclPro from the Web

You can obtain TclPro from Scriptics' Web site. To install TclPro from the web, go to the TclPro download page at *http://www.scriptics.com/tclpro/eval*. Follow the instructions provided there to download the TclPro distribution and install it on your system.

Installing Adobe Acrobat Reader

Note You need Acrobat Reader 3.0+ to view this guide in PDF format. If you already have Acrobat Reader installed or do not wish to view the *TclPro User's Guide* on screen, you can skip this step.

To install Acrobat Reader from the TclPro CD, run the TclPro installation program as described in "Installing TclPro from CD" select the Acrobat Reader package when prompted.

To install Acrobat Reader from Scriptics' Web site, go to *ftp://ftp.scriptics.com/pub/tclpro/adobe*.

Entering TclPro License Information

You need a license to use any TclPro application. The TclPro installer prompts you for license information during installation. You can change your license information afterwards by running the TclPro License Manager:

- On a Windows system, select TclPro License Manager from the Start menu or run the **prolicense.exe** file, which is contained in the *win32-ix86\bin* subdirectory of the TclPro installation directory.
- On a Unix system, to run the graphical version, run **prolicense**, which is contained in the platform-specific *bin* subdirectory of the TclPro installation directory (*solaris-sparc/bin* for Solaris, *linux-ix86/bin* for Linux, *hpux-parisc/bin* for HP-UX, and *irix-mips/bin* for IRIX).
- On a Unix system, to run the command-line version, run **prolicense.tty**, which is contained in the platform-specific *bin* subdirectory of the TclPro installation directory (*solaris-sparc/bin* for Solaris, *linux-ix86/bin* for Linux, *hpux-parisc/bin* for HP-UX, and *irix-mips/bin* for IRIX).

You must provide your name and a license key to TclPro License Manager. You can find your license key:

- on your CD-ROM case
- on the packing list included with your TclPro shipment
- in an email sent to you after you purchase TclPro or download an evaluation copy

If you do not have a valid license, you may purchase one from *http://www.scriptics.com/buy* or obtain a free 15-day evaluation license from *http://www.scriptics.com/tclpro/eval*.

About the TclPro Documentation

TclPro documentation consists of the following:

- This guide in print, PDF, WinHelp (Windows), and HTML (Unix) formats
- *Getting Started with TclPro* in print, PDF, WinHelp (Windows), and HTML (Unix) formats

- *TclPro Installation Guide* in print format
- Tcl and Tk command and C API reference pages in WinHelp (Windows) and HTML (Unix) formats

In this guide, Tcl commands, shell commands, and C functions appear in **bold** format. Variables, file names, and URLs appear in *italics*.

When this guide provides instructions for selecting an item from a menu, it lists the options you need to select separated by "|" characters, with the accelerator keys underlined. For example, "select File | Open from the menubar" means that you should click on the File menu in the application, then select the Open option from that menu; alternatively, you could hold the <Alt> key while you type "fo".

Tcl scripts, C programs, and computer output appear in a typewriter-style font. Information that you type at a Tcl or shell prompt is in a **bold typewriter-style font**. The following shows a simple example where you enter a Tcl command in **tclsh** and see the results:

```
% puts "2 + 2 = [expr 2 + 2]"
2 + 2 = 4
```

For More Information

This section lists sources of additional information about TclPro and Tcl/Tk.

TclPro Technical Support

Scriptics Corporation offers several levels of Technical Support. In addition to phone & email support for qualified customers, we also have online FAQs, a Known Bugs list, and other useful resources. For information for TclPro Technical Support, please see Scriptics' Web site: *http://www.scriptics.com/support*.

Finding Information about Tcl/Tk on the Web

Web sites that provide information about Tcl/Tk include:

- The Tcl Resource Center provides an annotated index to Tcl-related Web sites to help you find the information that you are seeking. See the Tcl Resource Center at *http://www.scriptics.com/resource*.
- The Tcl/Tk Consortium is a nonprofit organization dedicated to promoting Tcl/Tk. See their Web site at *http://www.tclconsortium.org*.

The *comp.lang.tcl* newsgroup provides a forum for questions and answers about Tcl. announcements about Tcl extensions and applications are posted to the *comp.lang.tcl.announce* newsgroup.

Tcl/Tk Training

Scriptics offers both public and on-site technical training courses for novice and advanced Tcl/Tk developers interested in harnessing the power of scripting. Our introductory tutorials bring novice Tcl/Tk programmers the skills they need to start creating exciting applications. Our advanced courses improve your productivity, showing how to create network applications with improved graphical features, and how to use object-oriented techniques with [incr Tcl]. Scriptics instructors also spend time showing how to link Tcl with your existing code base, and how to package your Tcl code in reusable libraries.

For our current training schedule and complete course descriptions, see our training Web page: *http://www.scriptics.com/training*

Related Documentation

If you are new to Tcl/Tk, here are some programming guides that can help you get started:

- *Practical Programming in Tcl and Tk*, by Brent Welch, published by Prentice Hall, 1997.
- *Graphic Applications for Tcl/Tk*, by Eric F. Johnson, M&T Books, 1997.
- *Effective Tcl/Tk Programming; Writing Better Programs with Tcl/Tk*, by Mark Harrison and Michael McLennan, published by Addison Wesley, 1998.
- *Tcl/Tk for Real Programmers*, by Clifton Flynt, published by Academic Press Professional, 1998.
- *Tcl/Tk for Programmers With Solved Exercises That Work With Unix and Windows*, by J. A. Zimmer, published by IEEE, 1998.
- *Tcl and the Tk Toolkit*, by John Ousterhout, published by Addison-Wesley, 1994.

For a comprehensive list of books related to Tcl/Tk, browse the Tcl Resource Center: *http://www.scriptics.com/resource*.

Chapter 2
TclPro Interpreters and Extensions

In addition to various Tcl development applications, TclPro is an extended Tcl platform that includes several popular Tcl extensions and enhanced Tcl interpreters.

TclPro Interpreters

The TclPro installation includes two enhanced Tcl interpreters, **protclsh82** and **prowish82**. These interpreters are identical to the standard **tclsh** and **wish** interpreters that are part of the Tcl and Tk distributions except for three improvements:

- **protclsh82** and **prowish82** are precompiled for all of the TclPro supported platforms. You don't need to compile them from source files.
- **protclsh82** and **prowish82** support all the extensions included with TclPro, as discussed in "Extensions Incorporated in TclPro" on page 8.
- **protclsh82** and **prowish82** support an extension called **tbcload**. This extension is required to run the bytecode files created by TclPro Compiler.

Running the TclPro Interpreters on Unix

To simplify running **protclsh82** and **prowish82** on Unix systems:

1) Add the TclPro *bin* directory to your *PATH* environment variable.

   This is a platform-specific directory in the install area of TclPro where all the executables are kept. It is *solaris-sparc/bin* for Solaris, *linux-ix86/bin* for Linux, *hpux-parisc/bin* for HP-UX, and *irix-mips/bin* for IRIX. For example, if TclPro was installed in */opt/TclPro*, you should add */opt/TclPro/solaris-sparc/bin* to the *PATH* environment variable on Solaris platforms.

2) If your scripts start with the following lines, they will be processed by **protclsh82** automatically:

```
#!/bin/sh
# the next line restarts using protclsh82 \
exec protclsh82 "$0" "$@"
```

You can modify this line to include **prowish82** or the interpreter of your choice. See the manual pages for **protclsh82** or **prowish82** for more information.

Running the TclPro Interpreters on Windows

If you are using Windows, you do not need to modify your path; the TclPro installer does this automatically. The TclPro installer also registers **prowish82** to handle files with the *.tcl* extension.

Extensions Incorporated in TclPro

TclPro incorporates several widely-used Tcl extensions with its distribution. Beyond simply providing source code for the extensions, TclPro integrates the extensions in several ways to make it easier for you to use the extensions in your applications:

- Each extension is pre-compiled for all of the TclPro supported platforms. You don't need to compile them from source files. If you are writing a custom interpreter, Chapter 7, "Creating Custom Interpreters with TclPro" on page 107 describes the locations of the extension libraries and provides information about linking with them.
- The enhanced TclPro interpreters, **protclsh82** and **prowish82**, provide built-in support for all incorporated extensions.
- TclPro Checker and TclPro Debugger understand all new commands and control structures implemented by each extension.
- TclPro Wrapper provides built-in support for creating wrapped applications that use the bundled extensions.

The following sections describe each of the extensions bundled with TclPro.

Important Each extension traditionally provided its own custom interpreter with built-in support for that extension's commands and control structures (for example, **itclsh** or **expect**). With TclPro, support for these extensions is provided by the **protclsh82** and **prowish82** interpreters. To use the features of a particular extension, you must

execute an appropriate package require command before using any of the commands or control structures of that extension. Table 1 lists the necessary **package require** command for each supported extension.

Table 1 Required Commands for Loading Supported TclPro Extensions

Extension	Required Initialization Command
Expect	**package require Expect**
[incr Tcl]	**package require Itcl**
[incr Tk]	**package require Itk**
[incr Widgets]	**package require Iwidgets**
TclX	**package require Tclx**
TkX	**package require Tkx**

[incr Tcl]

[incr Tcl] adds object-oriented programming support to Tcl. [incr Tcl] allows you to create *objects* in Tcl scripts, which act as building blocks for an application. Each object can contain its own data and procedures for manipulating the object. Objects are organized into *classes* with identical characteristics, and classes can inherit functionality from one another.

[incr Tcl] includes the [incr Widgets] library of more than 50 “mega-widgets,” including a combo-box, a tabbed notebook, a calendar, and an HTML viewer. These widgets work just like the usual Tk widgets, so you can use them even if you don't know anything about object-oriented programming. [incr Tcl] also comes with the [incr Tk] framework for creating your own mega-widget classes.

For more information about [incr Tcl], visit the web site *http://www.tcltk.com/itcl*.

Expect

Expect is a tool for automating interactive applications that have a command-line interface. Expect makes it easy to automate repetitive tasks or to add a graphical interface to an existing command-line tool.

For more information about Expect, visit the web site *http://expect.nist.gov.* You can also refer to the book *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, by Don Libes, published by O'Reilly & Associates, 1994.

Note Expect command names that “collide” with command names in the Tcl/Tk core (for example, **send**) can confuse TclPro Checker when it checks an Expect script, causing it to report syntax errors. To avoid this, use the **exp_** prefix for all such ambiguous commands (for example, use **exp_send** instead of **send**).

Extended Tcl (TclX)

Extended Tcl (TclX) provides additional support for systems programming tasks and large application development. Features of TclX include:

- Enhanced file manipulation and scanning
- Additional list manipulation commands
- Additional math commands
- Additional string commands
- X/Open Portability Guide, Version 3 (XPG/3) message catalog support
- Extended Unix access
- A help facility

For more information about TclX, visit the web site *http://www.neosoft.com/tclx*.

Chapter 3
TclPro Debugger

TclPro Debugger provides a variety of features that help you to find and fix bugs in Tcl scripts quickly. These features include:

- Stepping functions for evaluating single Tcl commands or running to where you have placed the cursor in the code
- Display of variable values for all accessible stack frames
- Full stack information and navigation around the stack and source code when the application is stopped
- Line- and variable-based breakpoints
- An Eval Console in which you can enter code for the application to evaluate dynamically when the application is stopped
- The ability to interrupt code to determine the execution status of the application that you are debugging
- The ability to communicate with remote and embedded applications

Overview of TclPro Debugger

This section lists the platforms and Tcl versions that TclPro Debugger supports. It then describes how to start TclPro Debugger and provides a tour of the TclPro Debugger main window.

Supported Tcl Versions

TclPro Debugger can debug any Tcl/Tk script running in a Tcl version 7.6 and Tk version 4.2 or later interpreter. This includes any extensions to those interpreters that do not radically redefine any standard Tcl commands.

Important Renaming or radically redefining any standard Tcl commands may cause TclPro Debugger to fail. An example of a radical redefinition of the **proc** command would be to redefine it to take four arguments instead of three. In particular, avoid altering the Tcl commands listed below:

array	break	catch	cd
close	concat	continue	eof
error	eval	event	exit
expr	fconfigure	file	fileevent
flush	for	foreach	gets
global	if	incr	info
lappend	lindex	linsert	list
llength	lrange	lreplace	lsearch
namespace	open	proc	puts
pwd	read	regexp	regsub
rename	return	set	string
switch	trace	unset	uplevel
upvar	variable	vwait	while

Starting TclPro Debugger

If you are running TclPro Debugger on a Windows system, select TclPro Debugger from the Start menu or double-click the *prodebug.exe* icon. If you are running Unix, add the release directory to your execution path, and enter **prodebug** at the prompt.

The TclPro Debugger Main Window

Figure 1 shows the main window that TclPro Debugger displays when it starts. The main window includes the following subregions:

- Tool bar
- Stack display
- Variable display
- Code display
- Result display

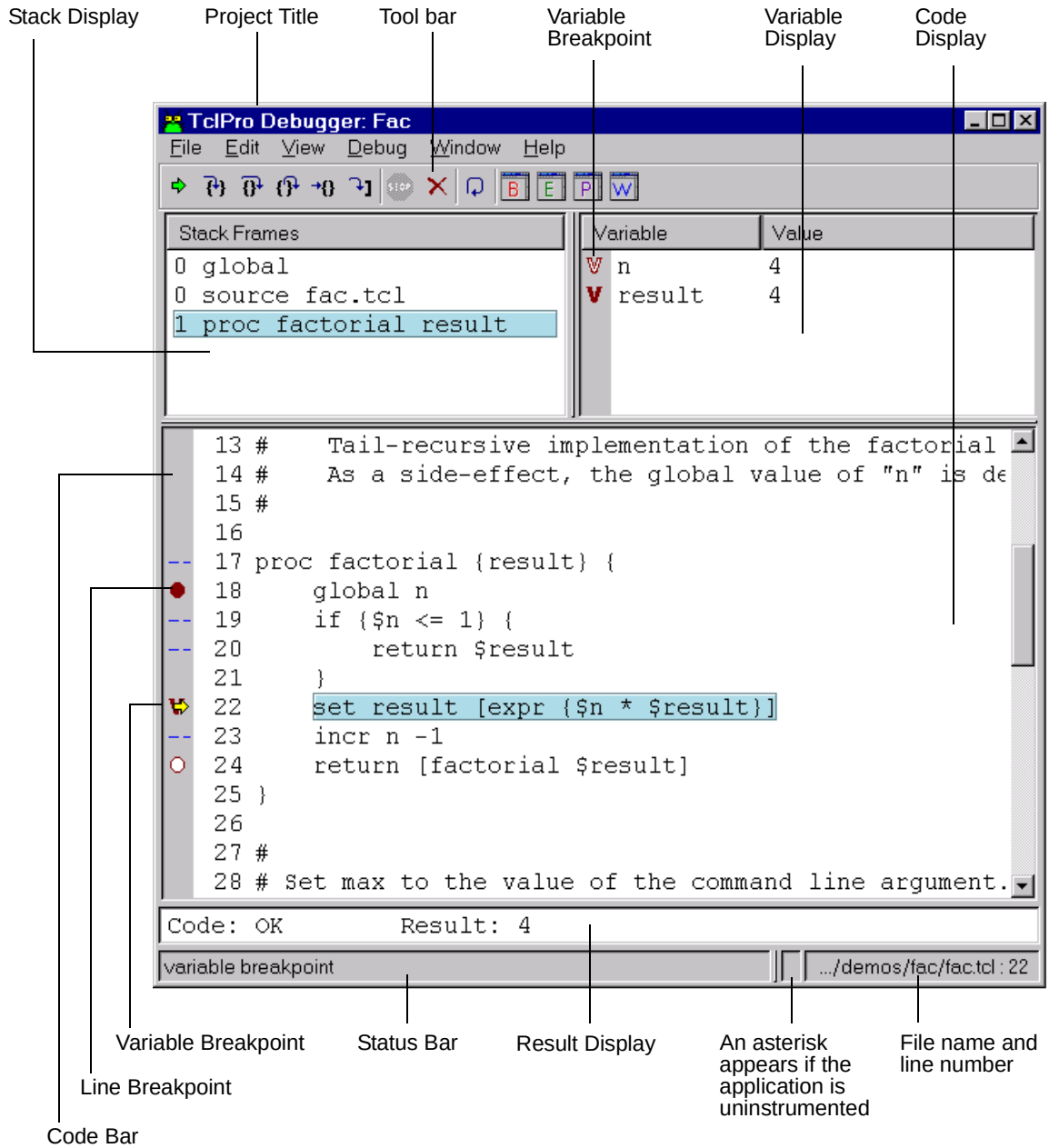


Figure 1 TclPro Debugger Main Window

The main window, as shown in Figure 1, includes menus and a tool bar, in which you run, step through, interrupt, kill, or restart your code. You can change the appearance of TclPro Debugger by toggling the display of various elements of the Main window:

Tool bar Select View | Toolbar from the menubar

Results display
Select View | Result from the menubar

Status bar Select View | Status from the menubar

Line numbers
Select View | Line Numbers from the menubar

The Tool Bar

Figure 2 shows the tool bar, with callouts identifying each of the buttons. The function of each button is described in the following sections.

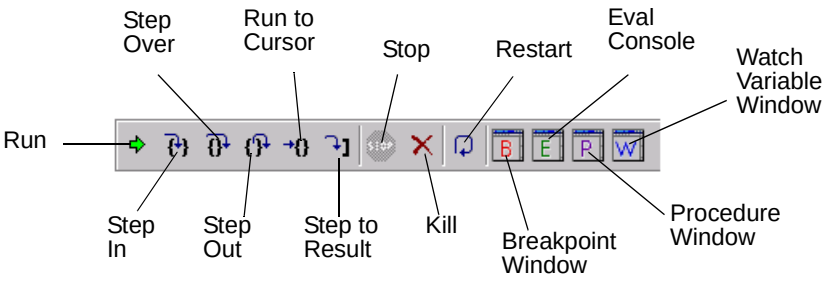


Figure 2 TclPro Debugger Tool Bar

When you hold your mouse over a tool bar button, a description of the functionality of that button appears in the left side of the status bar.

The Stack Display

The Stack Display shows the most recent stack levels and highlights the current location in your code when the application is stopped. If you select a stack level, TclPro Debugger shows the code and variable values for that stack level in the code display and the variable display. When the application encounters a breakpoint, the last stack frame is automatically selected and highlighted in the Stack Frame display. The call stack includes an entry for each distinct scope or body of code. It displays stack frame information in this format: stack level, Tcl command, and

relevant arguments. Stack level 0 indicates the global level. Stack level 1 indicates that a procedure is invoked from level 0; stack level 2 indicates that a procedure is invoked from stack level 1, and so on.

Note If your code is in an event loop when you click the Stop button, no code is shown in the Code display and the top level in the stack frame displays "event."

The following example shows a sample stack frame:

```
0 global
0 source myScriptFile
1 proc myProc arg1 arg2 arg3
2 namespace eval myNamespace
3 proc myproc3
0 uplevel
1 proc myproc3 args
```

In this example, the stack level is reset to 0 by the **uplevel** command; the **uplevel** command can be called explicitly in your source code or implicitly by a callback. As with any other procedure call, the **namespace eval** command creates a new level.

You can navigate through the application by clicking on specific stack frames, which affects both the Variable and Code displays. When you double-click any part of a stack frame, the Code display scrolls to and highlights the current command in that stack frame. For example, if you want to see the code that caused a stack frame to be created, you can double-click the frame directly above the frame in question. In addition to highlighting the current command, if the last stack frame is selected, TclPro Debugger indicates the current command with a yellow Run Arrow in the Code bar. TclPro Debugger also indicates the current command with a green triangular History Arrow in the Code bar. When you click a stack frame, the Variable display shows the variables in that stack frame. For example: if you want to see global variables, you can double-click any Level 0 stack frame. If you click directly on an argument in a **proc** stack frame, the Variable Window scrolls to and highlights the selected argument.

The Variable Display

The Variable display shows all of the existing variables in the highlighted stack frame. The value of each variable is updated whenever the application is stopped. The value for each array appears as an ellipsis (...). You can expand and contract the display of the array by clicking the ellipsis. When an array is expanded, each index is listed with its corresponding value. You can click to the left of the name of the variable to set a variable breakpoint, which causes the application to stop whenever the chosen variable is modified. See "Manipulating Breakpoints" on

page 35. Large variables are abbreviated in the Variable display. To see the complete value, double-click the variable, and the Data Display window appears. See “Displaying Data” on page 41.

If the message “No variable info for this stack” appears in the Variable display, it means that the stack level that is highlighted in the Stack display is hidden. Stack levels are hidden as a result of calls to Tcl commands like **vwait** and **uplevel**. When **vwait** is called, it creates a new stack, and all of the non-zero levels of the old stack are hidden until the **vwait** call returns. When **uplevel** is called with the absolute level for *x*, all of the levels of the old stack that are greater than *x* are hidden until the **uplevel** call returns.

The Code Display

The Code display shows exactly one Tcl code source at a time. A code source is either a file opened in the File menu, a file that has been sourced by the application, or a chunk of code dynamically created at runtime by commands such as **eval**. The Window menu lists all the open files, allowing you to select the file you want to view. You can also select a code source to view from the Breakpoint and Procedures windows. See “Using Breakpoints” on page 34 and “Finding Procedures” on page 38.

When the application is stopped, an arrow or triangle appears in the code bar indicating the current command with highlighted text. For example, in Figure 1 on page 13, the portion of the code that is highlighted is code that is about to be executed and it is also indicated by the yellow run arrow in the code bar. Code is also highlighted if it is found using the Find command. See “Going to a Specified Line” on page 37 for information on commands that you can use to move through and search for specific portions of code.

The main window includes a status bar. The left portion of the status bar displays the information about the state of the debugger session, or information about the tool bar buttons if you place your cursor over a button. The center displays an asterisk (“*”) if the current code source is uninstrumented; see “About TclPro Instrumentation” on page 50. The right portion displays the current file name and line number.

If you see the message “No Source Code...” in the Code display, there are two possible reasons:

- If your application is in the event loop when you click the Stop button, TclPro Debugger cannot display code because no code is being evaluated.
- TclPro Debugger cannot display code for the first stack level labeled “global” because your application’s code is sourced by code internal to TclPro Debugger.

The Result Display

The Result display shows the result and completion code of the most recently executed Tcl command. The Result display is not a scrolling window; TclPro Debugger displays only as much of the result as can fit in the Result display. You can double-click on the result to display it in the Data Display window (see “Displaying Data” on page 41).

Note The performance of TclPro Debugger can decrease if your application produces particularly long results (for example, reading a large file into a variable) and you have the Result display visible. If you want to increase performance in cases like this, toggle off the Result display by selecting View | Result from the menubar.

Managing Projects

You can manage multiple *projects* with TclPro Debugger. TclPro Debugger saves project information in files with the *.tpj* extension. Projects store a variety of information about an application including:

- the name of the initial Tcl script
- the interpreter
- any command-line arguments you pass to the script
- the current working directory
- any line breakpoints you have set
- any variables in your watch list

By default, when TclPro Debugger starts, it automatically reloads the last project you had open. You can change this behavior as described in “Startup and Exit Preferences” on page 48.

Note You must have a project open to perform any debugging actions.

Creating a New Project

To create a new project:

1) Select File | New Project from the menubar. If you have a project already open, TclPro Debugger prompts you to save that project.

   TclPro Debugger then opens the Project window shown in Figure 3.

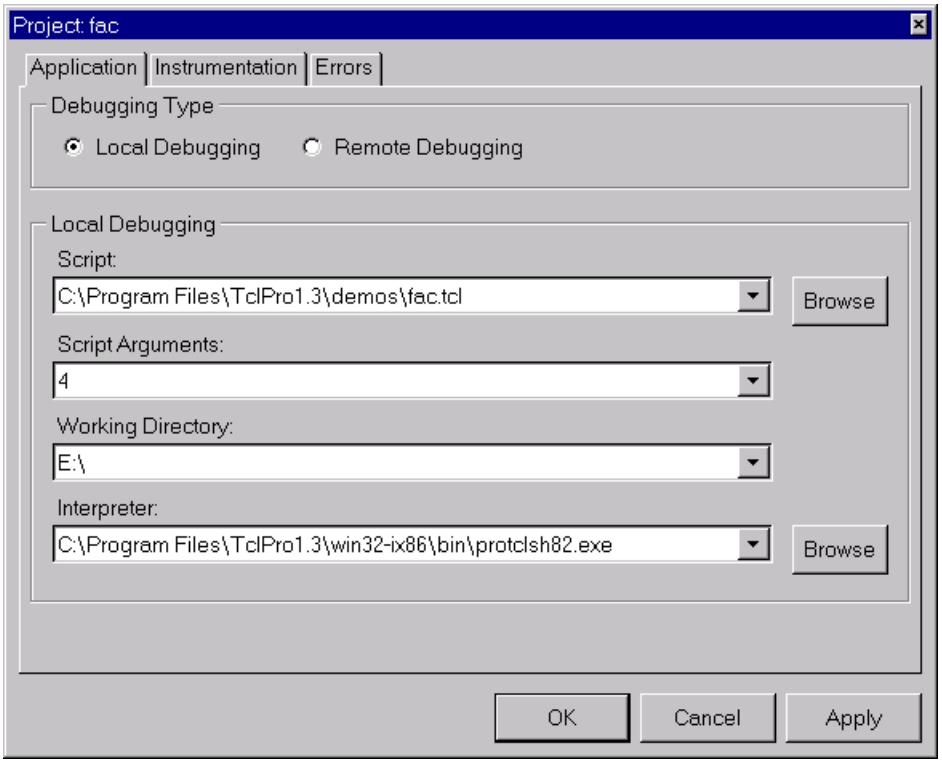


Figure 3 The TclPro Debugger Project Window

2) Select Local Debugging to debug a Tcl script running normally on your system. Select Remote Debugging only to debug a remote, embedded, or CGI Tcl application. See “Debugging Remote, Embedded, and CGI Applications” on page 51 for information on remote debugging.
3) In the Script field, type the path and name of the Tcl script to run, or click the Browse button next to the field to locate the Tcl script.
4) (Optional) In the Script Arguments field, type any script arguments you want to pass to the script when you run it under the debugger.
5) (Optional) In the Working Directory field, type the full path of the directory that you want to use for the working directory inside the Tcl/Tk script. If you don’t specify a working directory, TclPro Debugger uses the directory which contains the Tcl script you are debugging.

6) In the Interpreter field, type the path and name of the Tcl interpreter or click the Browse button next to the field to locate the interpreter. You can use any Tcl interpreter, such as **prowish82**, **protclsh82**, or a custom Tcl shell. You can also choose one from the drop-down list, which contains a list of Tcl interpreters set by your project defaults.

Note TclPro Debugger works properly with most custom Tcl interpreters. However, if your interpreter doesn't accept as its first command-line argument a Tcl script to execute or if it doesn't pass subsequent command-line arguments to the script using the standard *argc* and *argv* Tcl variables, then you must take special steps to use your interpreter with TclPro Debugger. See "Using Custom Tcl Interpreters with TclPro Debugger" on page 56 for more information.

Tip If there are one or more interpreters you commonly use, you can change your default project settings to include them in the Interpreter drop-down list:

a) Bring up the default project settings, as described in "Setting Default Project Settings" on page 27.

b) Type the path and name of the Tcl interpreter or click the Browse button next to the field to locate the first interpreter you want to appear in the drop-down list.

c) Repeat b) for each interpreter that you want to add to the list.

d) Save your default project settings.

The interpreters you specify are now available for all new projects you create afterwards.

7) The Instrumentation and Errors tabs allow you to fine tune TclPro Debugger's control over your application as you debug it. See "Changing Project Settings" on page 20 for information on these tabs.

8) Click the OK button to apply your choices and close the Project window, the Cancel button to cancel your choices and not open the new project, or the Apply button to apply your choices and keep the Project window open.

Once you create your new project, TclPro Debugger displays the Tcl script file you specified in the Code display of the main window. TclPro Debugger does not run the script until you tell it to do so, as described in "Controlling your Application" on page 29.

Opening an Existing Project

There are two ways that you can open an existing project in TclPro Debugger:

- Select File | Open Project from the menubar and select the project file you want to open from the file browser displayed.
- Select File | Recent Projects and select the project file you want to open.

If you already have a project open, TclPro Debugger first asks you whether or not you want to save that project before opening the project you select.

When you open an existing project, TclPro Debugger restores all of the project settings and breakpoints in effect when you saved the project. TclPro Debugger also displays the Tcl script file that you were viewing when you saved the project.

Saving a Project

To save a project, select File | Save Project from the menubar. The first time you save a project, specify the file name and location for your project. TclPro Debugger saves your project settings and any breakpoints and any watch variables you have set.

To save a project with a different name, elect File | SaveAs Project from the menubar.

Closing a Project

To close a project, select File | Close Project from the menubar. If you made changes, TclPro Debugger asks you whether or not you want to save the project before closing it.

Closing a project closes the project file and clears all the TclPro Debugger displays.

Changing Project Settings

To change the settings of the currently open project, select File | Project Settings from the menubar. TclPro Debugger displays the Project window shown in Figure 3 on page 18. From this window you can change the script, interpreter, instrumentation, and error settings for the project as described in the sections below.

Note Changes that you apply to your project settings while your application is running don't take effect until the next time you restart your application.

Changing Project Application Settings

The Application tab of the Project window lets you select basic application settings such as the Tcl script to debug and the Tcl interpreter to use. The contents of the Application tab depend on the Debugging Type option you select:

Local Debugging
: Debug a Tcl script running normally on your system.

Remote Debugging
: Debug a remote, embedded, or CGI Tcl application. See “Debugging Remote, Embedded, and CGI Applications” on page 51 for information on remote debugging.

If you select the Local Debugging option, the Application tab appears as shown in Figure 4.

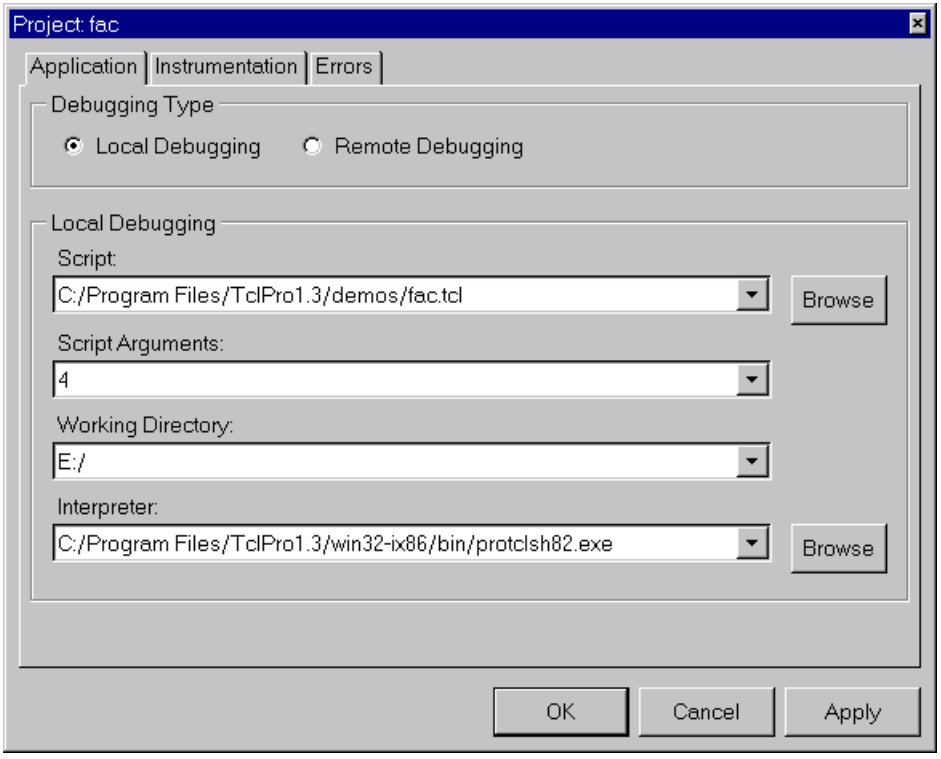


Figure 4 The Project Application Settings Tab, Local Debugging

You can change the following Local Debugging settings for a project:

Script — Type the pathname of the Tcl script to run, or click the Browse button next to the field to locate the Tcl script. You can also select the script from the drop-down list, which lists scripts that you have used recently in this project.

Script Arguments — Type any script arguments you want to pass to the script when you run it under the debugger. You can also select the arguments from the drop-down list, which lists the arguments that you have specified recently in this project.

Working Directory — Type the full path of the directory that you want to use for the working directory inside the Tcl/Tk script. If you don't specify a working directory, TclPro Debugger uses the directory that contains the Tcl script you are debugging. You can also select the working directory from the drop-down list, which lists the working directories that you have used recently in this project.

Interpreter — Type the path and name of the Tcl interpreter or click the Browse button next to the field to locate the interpreter. You can use any Tcl interpreter, such as **prowish82**, **protclsh82**, or a custom Tcl shell. You can also choose one from the drop-down list, which contains Tcl interpreters that have been installed in the standard locations on your computer and any other Tcl interpreters that you have previously specified for this project.

Note TclPro Debugger works properly with most custom Tcl interpreters. However, if your interpreter doesn't accept as its first command-line argument a Tcl script to execute or if it doesn't pass subsequent command-line arguments to the script using the standard *argc* and *argv* Tcl variables, then you must take special steps to use your interpreter with TclPro Debugger. See "Using Custom Tcl Interpreters with TclPro Debugger" on page 56 for more information.

If you select the Remote Debugging option, the Application tab appears as shown in Figure 5.

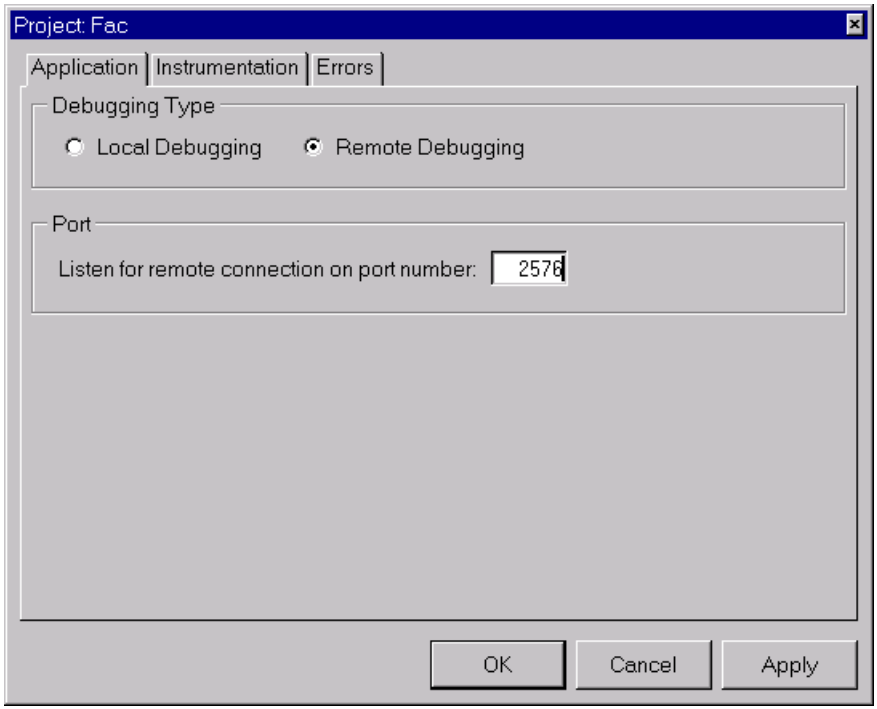


Figure 5 The Project Application Settings Tab, Remote Debugging

The only application setting you can change when debugging remotely is the TCP port that TclPro Debugger uses to communicate with the remote application. This is the port that you need to pass to **debugger_init** when starting your debugging session from a remote application. See “Debugging Remote, Embedded, and CGI Applications” on page 51 for information on remote debugging.

Note Changes that you apply to your project settings (by clicking either the OK or Apply button) while your application is running don't take effect until the next time you restart your application.

Changing Project Instrumentation Settings

The Instrumentation tab of the Project window, shown in Figure 6, lets you select files and classes of procedures that TclPro Debugger should and should not instrument. Instrumenting a file gives TclPro Debugger control over its execution, and allows you to set breakpoints, single-step through the file, and perform other debugging tasks. If a file is not instrumented, you can't perform debugging tasks

while your application is executing the file (or procedures defined in that file). For more information about instrumentation, see “About TclPro Instrumentation” on page 50.

Some cases of when you would want to control which files are instrumented and which files are not include:

- When you use a common Tcl script library for several projects in your organization. In this case, you would most likely debug the library separately and then instruct TclPro Debugger not to instrument the library when you later debug individual projects that use that library.
- When you debug large applications. Instrumenting a script takes time and slows the execution of your application. To minimize the overhead of debugging, it is more efficient to instrument and debug portions of your application separately.

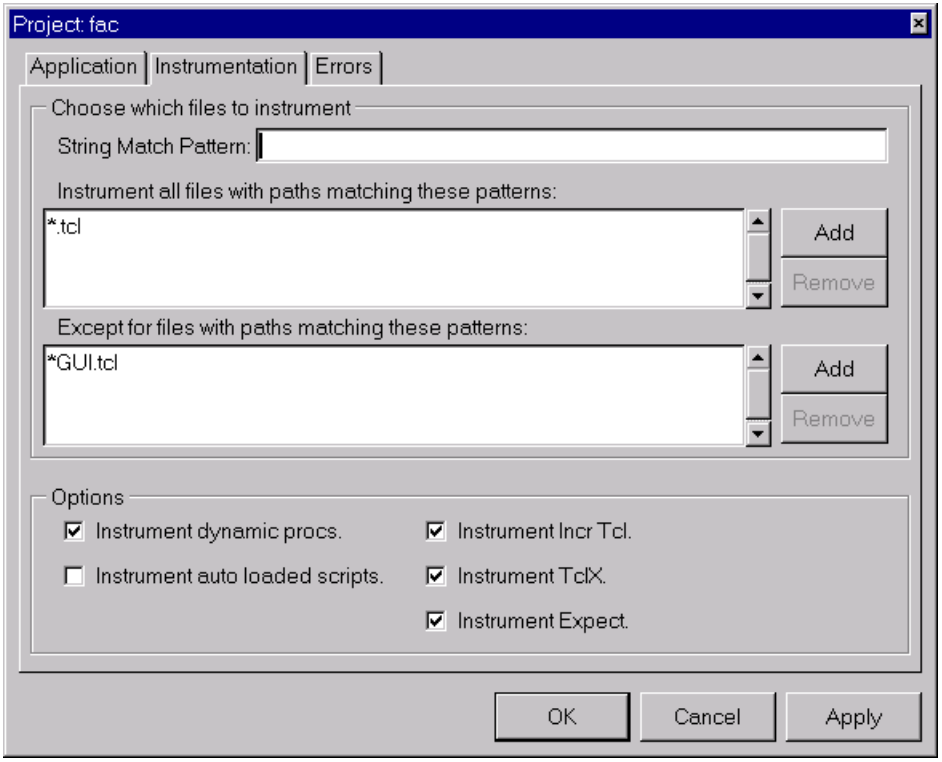


Figure 6 The Project Instrumentation Settings Tab

The top half of the Project Instrumentation dialog determines the files that TclPro Debugger instruments. (By default, all files are instrumented.) The first list box identifies a set of files to instrument, and the second list box identifies a subset of exceptions that are not instrumented. File name patterns follow the **string match** pattern conventions. (See the Tcl **string** command reference page for more information on pattern syntax.) Whenever your application **source**s a script file, TclPro Debugger compares the file name against the patterns you specify in this dialog to determine whether or not to instrument it. For example, setting the pattern "app*.tcl" in the first list box and "*GUI.tcl" in the second list box causes TclPro Debugger to instrument files such as *appMain.tcl* and *appStats.tcl*, but not instrument a file named *appGUI.tcl*.

To add a pattern to a list box, type the pattern in the String Match Pattern field and then click the Add button next to the appropriate list box. To remove a pattern from the list, click the file or pattern to highlight it, then click the Remove button.

Note If you delete all patterns in the first list box and then apply the setting (by clicking either the OK or Apply button), TclPro Debugger automatically adds the pattern "*" to the first list box. If TclPro Debugger didn't do this, then you could accidentally cause TclPro Debugger not to instrument any files, in which case you couldn't control your application with the debugger.

The lower half of the Project Instrumentation dialog provides finer control of the instrumentation of procedures and control structures in a script file:

Instrument Dynamic Procs
: Instrument procedures that you create dynamically. For example, selecting this check box instruments procedures created by the **eval** command.

Instrument Auto Loaded Scripts
: Automatically instrument auto-loaded scripts. You might want to turn this option off if you are using only standard Tcl extensions.

Instrument [incr Tcl]
: Instrument all your [incr Tcl] classes and methods.

Instrument TclX
: Instrument control structures in the TclX package, such as the **loop** command.

Instrument Expect
: Instrument the control structures in the Expect package, such as the **expect** command.

Note Changes that you apply to your project settings (by clicking either the OK or Apply button) while your application is running don't take effect until the next time you restart your application.

Changing Project Error Settings

The Errors tab of the Project window, shown in Figure 7, lets you specify how TclPro Debugger handles errors in your Tcl script:

Always Stop on Errors
: TclPro Debugger notifies you each time it encounters an error in the script (TclPro stops execution of your script even if your script catches the error)

Only Stop on Uncaught Errors
: TclPro Debugger notifies you only when it encounters an error not caught by the script (TclPro stops execution of your script only if your script does not catch the error)

Never Stop on Errors
: TclPro Debugger does not notify you about any errors in the application

Note For more information on how TclPro Debugger handles errors, see "Error Handling" on page 43.

Note Changes that you apply to your project settings (by clicking either the OK or Apply button) while your application is running don't take effect until the next time you restart your application.

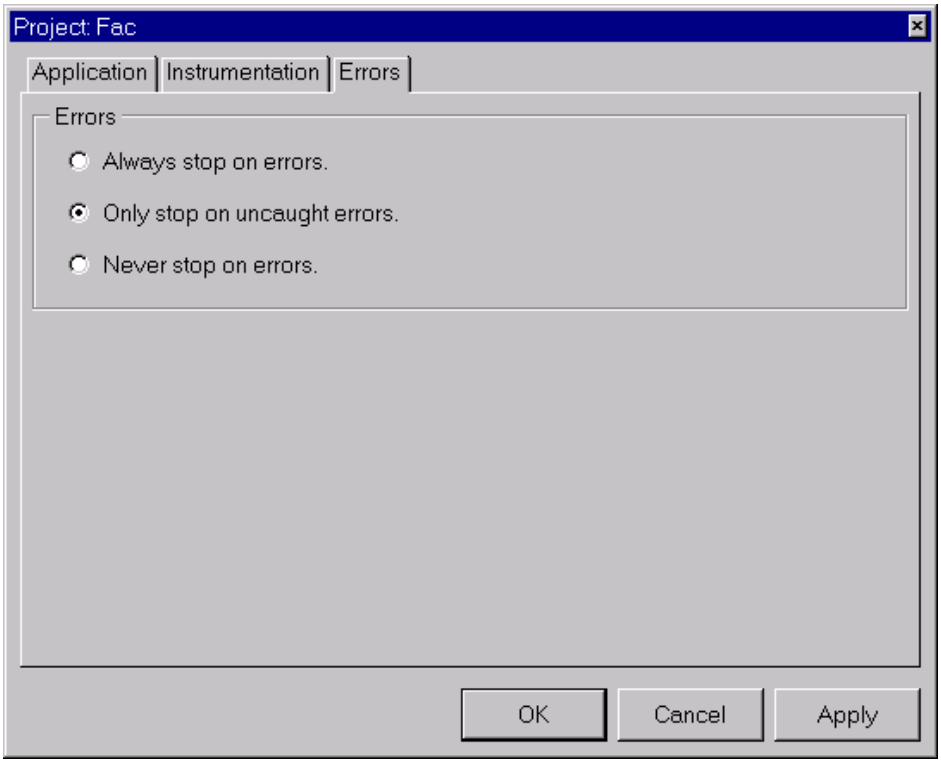


Figure 7 The Project Errors Settings Tab

Setting Default Project Settings

You can change any of the default project settings so that new projects you create have those settings. Changing the default project settings doesn't affect any existing projects you might have.

For example, if you commonly use a set of packages that you don't want TclPro Debugger to instrument, you could set those files in the default project settings. Then, any new project you create would pick up those instrumentation settings by default.

To set the default project settings:

1) If you have a project already open, select File | Close Project from the menubar to close that project.
2) Select File | Default Project Settings from the menubar. (TclPro Debugger displays this option only if you have no projects currently open.)

TclPro Debugger displays the Default Project Settings window. This window has the same tabs and settings available as in the Project window.

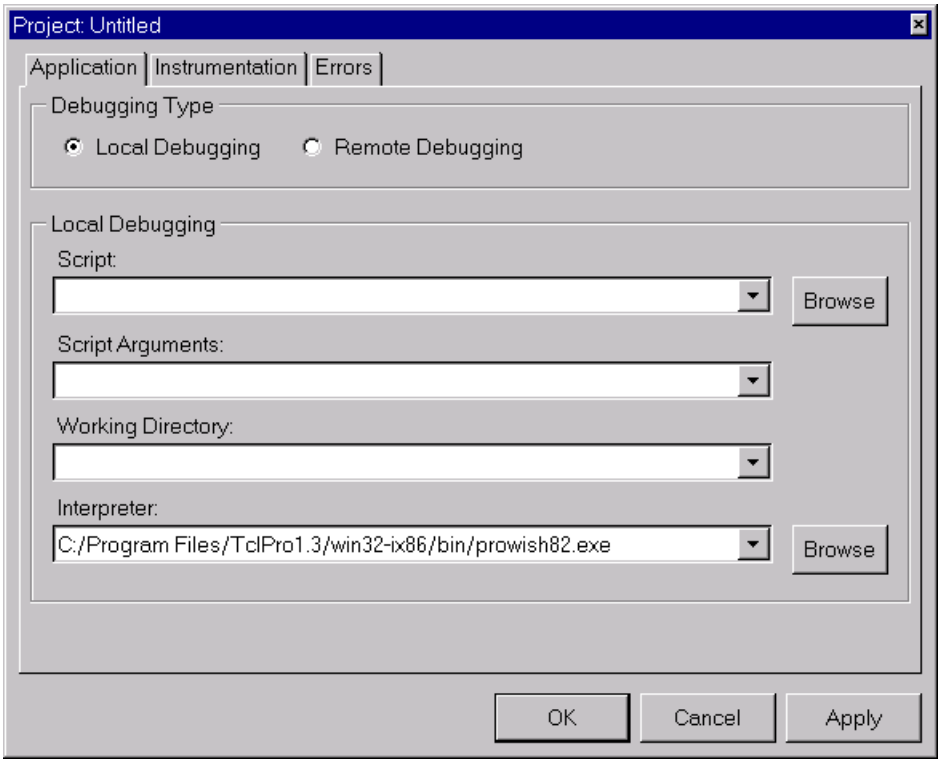


Figure 8 The Default Project Settings Window

3) Set the default project settings just as you set an individual project's settings. See "Changing Project Settings" on page 20 for a description of all project settings.

4) After changing the default project settings, click the OK button to save your choices and close the Default Project Settings window, the Cancel button to cancel your choices and close the Default Project Settings window, or the Apply button to apply your choices and keep the Default Project Settings window open

Opening a File

Opening a file in TclPro Debugger gives you the opportunity to create or check existing line-based breakpoints in the file before the file is sourced in the application. Breakpoints cause the application to stop before a line of code is executed so that you can examine the state of the application that you are debugging. See “Using Breakpoints” on page 34.

To open a file:

1) Select File | Open File from the menubar.
2) Type the full path and name of the file or browse and click the file that you want to open.

   The file that you opened appears in TclPro Debugger. You can view it using the scroll bars and menus.

Tip You can open a file at any time, even when an application is already running. When you open a file, TclPro Debugger reloads the file if the file has not been sourced by the running application or if no application is running. If the application is running and has sourced the file, modifications to that file cannot be seen in the Code display until that file is sourced again or the file is reopened after the application is terminated.

Controlling your Application

This section describes how to use the basic features of TclPro Debugger.

Running Code with TclPro Debugger

Click the Run button to run your code with TclPro Debugger, as shown in Figure 2 on page 14. When the application stops, TclPro Debugger indicates the line of code that it is processing with an arrow and highlights the portion of the line that it is about to execute.

Once the application is running, it stops at these events:

- At breakpoints. For information about breakpoints, see “Using Breakpoints” on page 34.
- If an error is detected, TclPro Debugger stops on the line of code that generated the error, and the code that caused the error is highlighted. See “Error Handling” on page 43.
- If you click the Run to Cursor icon in the tool bar, the application runs to the line where you placed your cursor.

Run to Cursor

The Run to Cursor icon in the tool bar, as shown in Figure 2 on page 14, enables you to create a temporary breakpoint that is automatically removed the next time TclPro Debugger stops. When your application is stopped, you can move the cursor to the line of code where you want to stop, and press the Run To Cursor button.

Note If the application stops for any reason, such as encountering another breakpoint or reaching the line containing the cursor, the temporary breakpoint is removed. The operation of the Run to Cursor feature is similar to those of line-based breakpoints. If the cursor is not set, or if it is on a line that is either empty or contains only comments or curly braces, clicking the Run to Cursor button is equivalent to clicking the Run button. The application stops just before evaluating the first command on the line regardless of where you place the cursor on a line of code.

Stepping through Code

TclPro Debugger offers four ways of stepping through your scripts: Step In, Step Out, Step Over, and Step To Result. When your application is stopped, you can step from the current command, indicated by the yellow run arrow in the code bar. To use the stepping features, click the corresponding button on the tool bar when your application is stopped. See Figure 2, “TclPro Debugger Tool Bar” on page 14.

Note If the application stops for any reason, such as encountering an error or breakpoint, after any of the Step buttons is pressed, the step is considered to be completed.

Stepping In

The Step In feature provides the finest granularity at which you can stop and inspect your application. Stepping in causes the application to stop just before executing the next instrumented command. Stepping in is useful for following the control flow of your application as it sources files, calls procedures, and evaluates command substitutions.

For example, if your application is stopped on the command

```
myProc [incr x 5]
```

you can Step In and stop the application before it evaluates the subcommand **incr x 5**. You can Step In again to stop the application on the first line of code in the body of the **myProc** procedure.

The following list describes the rules of behavior for the Step In function:

- If the current command contains subcommands, the application stops just before evaluating the first subcommand.
- If the current command is a call to an instrumented procedure, and all subcommands, if any exist, have been evaluated, the application stops on the first line of code in the body of the procedure.
- If the current command is a call to the **source** command, and all subcommands, if any exist, have been evaluated, the application stops on the first line of code in the sourced file.
- If the current command is not a call to an instrumented procedure, and all subcommands, if any exist, have been evaluated, the application stops on the first instrumented command called by the current command.
- If the current command does not call any instrumented code, then the **Step In** function behaves like the **Step Over** function.

Stepping Out

Stepping out causes the application to stop before executing the next command after the current stack level or body of code returns. The Step Out feature is useful for backing out of code you are no longer interested in inspecting. For example: if you are stopped in the body of the **myProc** procedure in the following application

```
1 source someFile.tcl
2 myProc [incr x 5]
3 myNextProc $x
```

and you would like to progress to the **myNextProc $x** command, you can Step Out of the **myProc** procedure, and then Step In the **myNextProc** procedure.

The following list describes the rules of behavior for the Step Out function:

- If the current command is in the body of a procedure, the application stops before executing the next command after the procedure returns.
- If the current command is at the global level of a file that has been sourced, the application stops before executing the next command after the code in the sourced file is evaluated.

- If the current command is at the global level of the main script file, clicking the Step Out button behaves like clicking Run button.

Stepping Over

Stepping over causes the application to stop just before executing the next command after the current command in your application is fully executed. The Step Over feature is useful for following the application as it progresses through a body of code at the current stack level. For example, suppose you are stopped on line 1 in the following application

```
1 source someFile.tcl
2 set x 1
3 myProc [incr x 5]
4 puts $x
```

If you Step Over the **source** command, the application stops at **set x 1**. If you continue to click Step Over, **myProc [incr x 5]** becomes the new current command, followed by **puts $x**.

The following list describes the rules of behavior for the Step Over function:

- If the current command is a call to an instrumented procedure, the application stops before the executing the next command after the procedure returns.
- If the current command is a call to the **source** command, the application stops before the executing the next command after the code in the sourced file is evaluated.
- If the current command is the last one at the current stack level or in the current body of code, Step Over behaves like Step Out.

Stepping to Result

Stepping to Result executes the current command and stops execution. After using Step to Result, TclPro Debugger highlights the command just executed and displays the result and return code of that command in the Command Results display of the debugger main window.

The Step to Result feature is useful for examining the results of nested commands. For example, suppose you click Step In at line 3 in the following application:

```
1 source someFile.tcl
2 set x 1
3 myProc [incr x 5]
4 puts $x
```

If you click Step to Result, your application executes the subcommand and stops. You can then examine the result of the subcommand before continuing. By comparison, clicking Step In again at this point would execute the subcommand **[incr x 5]** and immediately Step In to **myProc**, and clicking Step Over would execute both the **[incr x 5]** subcommand and the call to **myProc** before stopping.

Interrupting the Application

Clicking the Stop button causes TclPro Debugger to interrupt the application while it is running. You can interrupt the application at any time; when you interrupt, an implicit breakpoint is added to the next command to be executed in the script. The application stops as it would at any other breakpoint, and you can then interact with the application.

Note If your code is in an event loop when you click the Stop button, no code is shown in the Code display and the top level in the stack frame displays "*event*."

Note If your application is executing uninstrumented Code or is in a long-running command, TclPro Debugger may not be able to stop the application immediately.

Killing the Application

Clicking the Kill button causes TclPro Debugger to end the application's process. When you kill the application that you are debugging, information about its state is no longer available. You can then restart the application or launch another application.

Note You cannot terminate remote applications using the Kill button. You can terminate a remote application by interrupting the application and typing "exit" in the Eval Console. See "Manipulating Data" on page 42.

Restarting the Application

Click the Restart button to terminate the current application and then restart the same application. This is equivalent to killing the application and immediately restarting it. When you restart an application, TclPro Debugger automatically reloads the main script. This is useful if you have modified the script to fix a bug and want to start the application over to test the change.

If you have modified files other than the main script and wish to set or change breakpoints in those files, you can open them by selecting File | Open File from the menubar rather than viewing the stale files from the Window menu.

Quitting TclPro Debugger

To quit TclPro Debugger, select File | Exit from the menubar or click the Close button in the TclPro main window.

Using Breakpoints

A breakpoint causes the application to stop so that you can examine its state. You can add breakpoints in an application at any time. Using breakpoints, you can obtain information, such as variables and their values, the current call stack, and valid procedure names. TclPro supports two types of breakpoints: *line-based* and *variable* breakpoints.

Line-based breakpoints

Line-based breakpoints enable you to specify a line of code where the application should stop. Line-based breakpoints cause TclPro Debugger to stop before executing each command and subcommand on the specified line. Line-based breakpoints are persistent across runs of the application and debugger sessions.

TclPro Debugger does not stop at line-based breakpoints that are set in uninstrumented lines of code, blank lines, comment lines, and lines that contain only curly braces. However, variable breakpoints can be triggered if the variable is modified in uninstrumented code. See “About TclPro Instrumentation” on page 50 for information.

Variable Breakpoints

Variable breakpoints cause the application to stop when the variable is modified. Variable-based breakpoints are not stored in the application after you close it, or when the variable is removed, unset, or goes out of scope, for example: a local variable in a procedure goes out of scope when the procedure returns.

Note The Variable breakpoints track the unique location where the variable is stored in memory rather than the name of the variable. You can not set a variable breakpoint until the variable exists in the application.

Manipulating Breakpoints

You can create breakpoints in the main Debugger window, as shown in Figure 1, “TclPro Debugger Main Window” on page 13. To set a line-based breakpoint, click the code bar in the left margin in the Code display. The line-based breakpoint appears as a small stop sign, and causes the application to stop just before the line is executed.

To create a Variable breakpoint, click the left margin in the Variable display adjacent to the variable. The breakpoint appears as a large “V” in the Variable display. The “V” also appears in the code bar of the Code display when the variable breakpoint is triggered causing the application to stop. The variable breakpoint triggers when the value of the variable changes. You can also create breakpoints in the Breakpoint window; see Figure 9, “The Breakpoints Window.”

To delete a breakpoint, click the breakpoint in the Code or Variable display.

Viewing Breakpoints in the Breakpoints Window

To display the Breakpoints window, click the “B” in the tool bar or by select View | Breakpoints from the menubar. The Breakpoints window displays line-based and variable breakpoints, as shown in Figure 9.

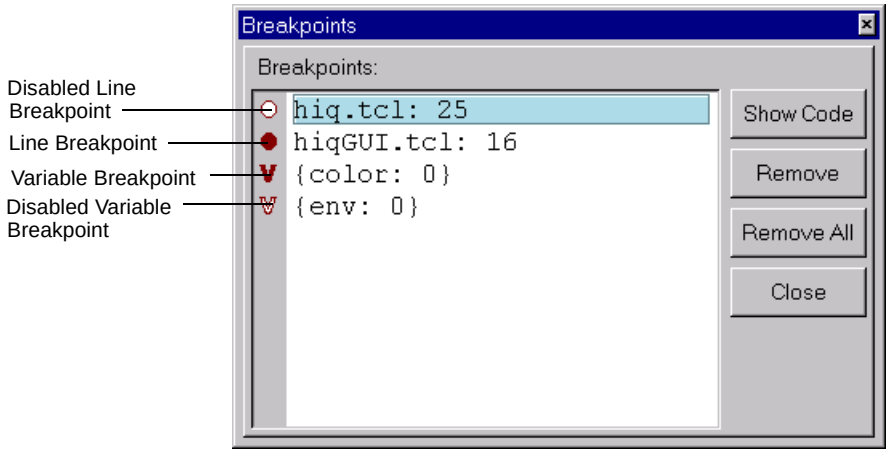


Figure 9 The Breakpoints Window

The line-based breakpoints in Figure 9 indicate the file and line number where the breakpoint has been set. To select a breakpoint, click the line to the right of the breakpoint in the Breakpoint window to highlight it. You can delete, disable, and enable breakpoints:

- To delete a breakpoint, select the line in the Breakpoint window and click the Remove button.
- To disable a breakpoint, click the breakpoint in the Breakpoint window.

  The disabled breakpoint is shown as a hollow stop sign for a line-based breakpoint or hollow “V” for a variable-based breakpoint.
- To enable a disabled breakpoint, click it in the Breakpoint window.

  Disabling and enabling breakpoints can be helpful when you want to keep all of your breakpoints but may not want to use all of them at the same time.

Note You can select multiple breakpoints to be disabled or enabled by clicking the breakpoints while pressing the <Ctrl> key.

You can perform the following actions on a selected breakpoint:

- Click the Show Code button to show the code at a Line-based breakpoint.

  Clicking this button causes TclPro Debugger to display the code containing the corresponding line in the Code display.
- Click the Remove button to remove a selected breakpoint.

You can click the Remove All button to remove all of the breakpoints.

The information for a variable breakpoint in the Breakpoint window, as shown in Figure 9, appears in the form of two sets. The first set contains the variable name followed by the absolute stack level at which the variable breakpoint was created. The second set contains information regarding the most recent occasion in which the variable breakpoint was triggered. If the second set is empty, the variable breakpoint has never been triggered. Otherwise, the second set contains the name and stack level of the variable that triggered the variable breakpoint. In most cases, the second set will not differ from the first set. However, when a variable is aliased by the **global** and **upvar** commands, any instance of that variable can trigger the variable breakpoint. The second set is helpful when you have an aliasing bug in your code.

The following is an example of an aliased variable *a* whose variable breakpoint gets triggered by a variable named *x*:

```
1 proc foo {} {
2     upvar #0 a x
3     set x 52
4 }
5 set a 50
6 puts "global var a is set"
7 set a 51
8 foo
```

If you stop this application on line 6, you can create a variable breakpoint for the global variable *a*. If you open the Breakpoint window, you will see the following:

```
{a: 0} {: }
```

If you continue to run the application, the variable breakpoint is triggered on line 7, the following appears in the Breakpoint window:

```
{a: 0} {a: 0}
```

If you continue to run the application again, the variable breakpoint is triggered once more on line 3, the following appears in the Breakpoint window:

```
{a: 0} {x: 1}
```

Navigating Code

TclPro Debugger provides utilities that help you can navigate to specific portions of the code that you are debugging, including Procedures window, the Goto command, the Find command, and the Window menu.

Going to a Specified Line

1) Select Edit | Goto Line from the menubar.
2) Type a line number in the text box.
3) Click the Goto Line button.

   TclPro Debugger highlights the specified line.

Tip You can also use the Goto What drop-down menu to move up or move down the lines in your code from the insertion cursor. Select Move Up Lines or Move Down Lines and type the number of lines that you want to move.

Using the Find Utility

1) Select Edit | Find from the menubar.
2) Type a code fragment or other string in the text box to locate that string. You can choose among several find options:
   - Select Match Whole Word only to find those strings that match the entire string that you typed. This option looks for white space as a delimiter, for example: if you searched for the string “sea” you would find all instances of “sea” but would not find “seashore”.
   - Select Match Case to find strings that match the case of the string that you typed. For example, with Match Case selected, searching for the string “sea” would not match “Sea”.

- Select Regular Expression to find strings that match the one you typed using the search format for regular expressions; see the **regexp** manual page for information. If you do not select this checkbox, it will perform searches that match all characters exactly.
- Select Search All Open Documents to find matching strings in all files that are currently open. The Window menu displays a list of all open files. If you don't select this options, TclPro Debugger searches only the current file (the one shown in the Code display).

3) Click the Direction for the search: Up or Down (default).

4) Press the <Enter> key to start the Find process.

   TclPro Debugger highlights the code that matches the string that you typed. If the string is not found, the Code Display does not change. You can find subsequent matching strings by clicking the Find again command or pressing the <F3> key.

Finding Procedures

You can use the Procedures window, shown in Figure 10, to view the list of procedures that have been defined in your application. To open the Procedures window, click the "P" button in the tool bar in the main TclPro Debugger window, or select View | Procedures from the menubar.

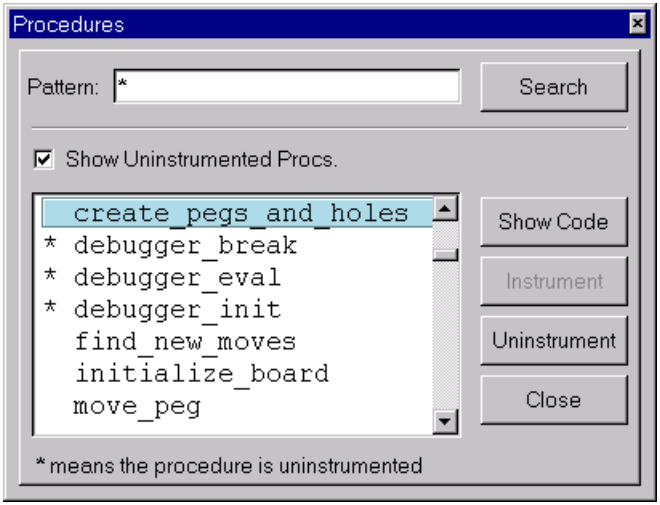


Figure 10 The Procedures Window

To narrow down the list, you can type a pattern in the text box and click Search. The default pattern is an asterisk ("*") which lists all of the defined procedures in the application. Pattern strings can be one or more characters and follow the search conventions that are used with the Tcl **glob** command. The matches for the string are shown in the body of the Procedures window. This is useful for finding specific procedures if you have large applications with many procedures. For example: if you type "tcl*" in the text box of the Procedures window shown in Figure 10, **tclLog**, **tclMacPkgSearch**, and all other procedures beginning with "tcl" are displayed in the display area of the Procedures window.

You can display both instrumented and uninstrumented procedures by selecting Show Uninstrumented Procs. TclPro Debugger indicates that a procedure is uninstrumented by listing the procedure preceded by an asterisk ("*") in the Procedures window. For more information about instrumentation, see "About TclPro Instrumentation" on page 50.

When you select a procedure from the list, you can perform any of the following actions on it:

Show Code
: Display the code where the procedure is defined, or the body of the procedure if the procedure is dynamically defined.

Instrument
: Instrument a selected procedure.

Uninstrument
: Uninstrument a selected procedure.

Using the Window Menu

Select the Window menu to display all of the files that are open in TclPro Debugger.

Displaying Code and Data

TclPro Debugger provides several windows in which you can display and monitor specific aspects of the application that you are debugging. These include the Watch Variable window, and the Data Display window. For information on the Breakpoints window, see "Viewing Breakpoints in the Breakpoints Window" on page 35.

Watching Variables

To open the Watch Variables window, click the “W” in the tool bar of the main window Select View | Watch Variables from the menubar. The Watch Variables window is shown in Figure 11 on page 40.

The Watch Variables window displays the variable names and their values at the stack level that is highlighted in the stack display. The values in the Watch Variables window are updated each time the application stops and also each time you select a new stack level in the Stack display in the main window. If a variable name is not defined at the selected stack level, then “<No Value>” appears instead of a value.

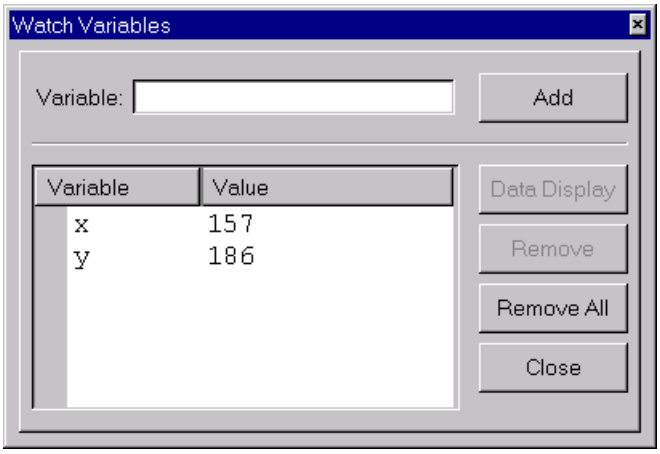


Figure 11 The Watch Variables Window

To add a variable name to the Watch Variables window:

1) Type the variable name in the text box of the Watch Variables window.
2) Click Add or press the <Return> key.

   The variable name and the current value of the variable are displayed in the large window.

   You can remove a specific variable name by selecting the line, and clicking the Remove button, or clicking Remove All to remove all the variables.

If you select a variable and click the Data Display button, the Data Display window appears.

The Watch Variables window is useful for observing variables in different stack levels that have the same name. For example: suppose the following script is stopped just before executing line 10:

```
1 proc bar {x} {
2 puts $x
3 }
4
5 proc foo {y} {
6 baz [expr {$y + 3}]
7 }
8
9 set x 2
10 foo $x
```

The stack display is shown below:

```
0 global
0 source sample.tcl
1 proc foo y
2 proc bar x
```

If you are watching the variable named *x*, you will see the value change as you select different stack levels. At level 2, *x* has the value 5. At level 1, *x* is not defined, so "<No Value>" is displayed. At level 0, *x* has the value 2.

Displaying Data

To open the Data Display window, double-click a variable in the Variable display in the main window or double-click a variable in the Watch Variable window, or select View | Data Display from the menubar. The Data Display window is shown in Figure 12 on page 42.

The Data Display allows you to see the full unabbreviated value of a variable, which can be helpful if you are looking at long strings.

There are two ways to change which variable is displayed in the Watch Variable window:

- Double-click a variable in either the Variable display or the Watch Variable window.
- Type the variable name in the text entry box and type <Return> or click the Inspect button.

The variable is linked to the stack level that is highlighted in the Stack display at the time the variable is entered in the Data Display window. Once the variable is entered, changing the stack level in the Stack display will not affect the value of the variable. The value that is displayed for the variable is updated each time the application stops. If "<No Value>" appears, it means that either the variable was unset or the stack level attached to the variable has returned. Like variable

breakpoints, a variable in the Data Display is associated with a location in memory. Once "<No Value>" appears, the previous memory location is no longer reserved for that particular variable, so "<No Value>" for the variable will reappear.

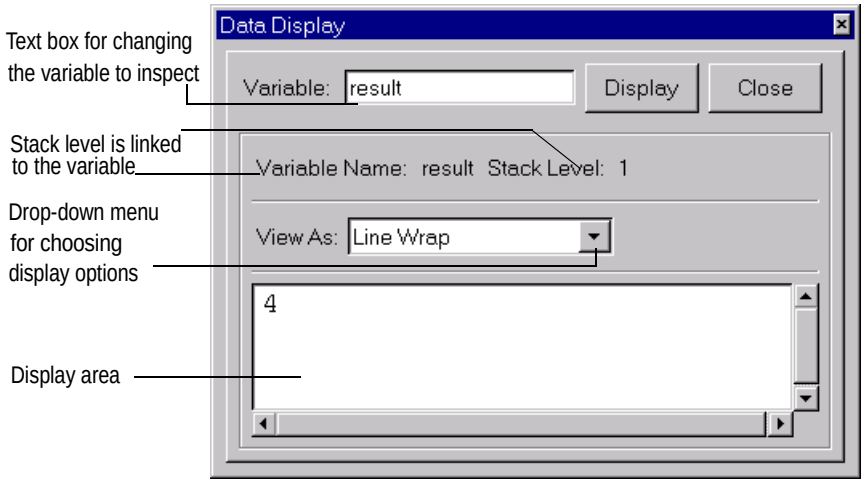


Figure 12 The Data Display window

Use the drop-down View As menu to select the format for the variables. TclPro Debugger attempts to match the display to the variable type, for example, if the variable is scalar, it will display with line wraps, and if it is an array, it will display as an array. You can view the variable with the following formats:

Line wrap	Wrap the line when it exceeds the length of the display window, which is the default display for scalar variables.
Raw data	Does not modify the display.
List	TclPro Debugger treats the variable value as a Tcl list, extracts the elements of the list, and displays each element on a separate line.
Array	Each element is displayed as a separate item with a name and value.

Note Ordered lists can be displayed as arrays.

Manipulating Data

To open the Eval console, click the "E" in the tool bar or select View | Eval Console from the menubar. The Eval console is shown in Figure 13 on page 43.

Using the Eval console, you can invoke commands in an application any time that the application is stopped. If you see something that is wrong or missing while debugging a program, you can type the missing information in the Eval console and it is immediately evaluated in the application.

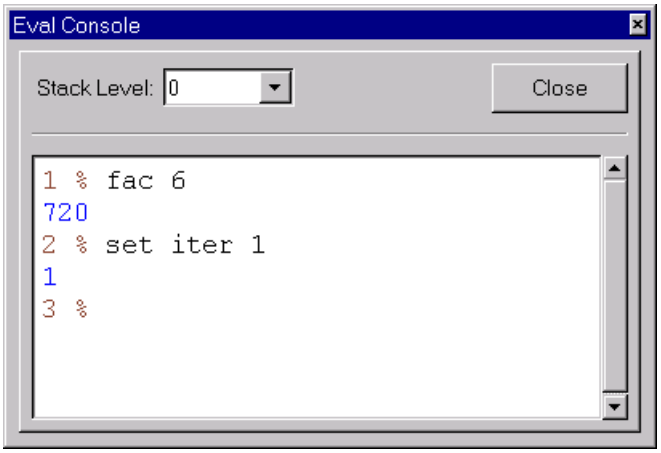


Figure 13 The Eval console

Using the Eval console, you can evaluate commands at any visible stack level. You can also call procedures from the Eval Console. You can choose among the available stack levels using the Stack Level drop-down arrow. Choosing the stack level is useful for setting global variables at level 0 and for calling procedures at various levels. When the Eval Console first appears, the default level is the deepest level in the stack display.

Tip You can also change the stack level in the Eval console by typing <Ctrl+Plus> to move to the next higher level stack frame or <Ctrl+Minus> to move to the next lower stack frame.

Error Handling

TclPro Debugger detects all errors in the application including runtime and parsing errors.

Parsing Error Handling

A *parsing error* is an error that is caused by code that is not syntactically valid. An example of a parsing error is a script that is missing a close brace. TclPro Debugger detects parsing errors during instrumentation, whenever a file is sourced or a procedure is created dynamically by the application.

When a parsing error occurs, TclPro Debugger cannot understand the script's control flow following the error, and cannot continue instrumenting the code. TclPro Debugger displays a dialog box in which you choose to either quit the application or continue the application with the partially instrumented script. If you choose to continue debugging the partially instrumented script, the same error appears as a runtime error if the code is executed. See "About TclPro Instrumentation" on page 50 for details on the implications of continuing despite the parsing error.

Runtime Error Handling

An example of a *runtime error* is an attempt to read a non-existent variable. TclPro Debugger detects all runtime errors, including both those caught and those not caught by a Tcl script. How TclPro Debugger handles runtime errors depends on the Error settings that you specify for your project. (See "Changing Project Error Settings" on page 26 for more information on specifying your project Error settings.) If you have set:

Always Stop on Errors
: TclPro Debugger notifies you each time it encounters an error in the script.

Only Stop on Uncaught Errors
: TclPro Debugger notifies you only when it encounters an error not caught by the script.

Never Stop on Errors
: TclPro Debugger does not notify you when it encounters errors in the application. Your application handles errors in the same manner as it would if it were not running under TclPro Debugger.

When TclPro Debugger detects a runtime error in accordance with the rules above, it stops execution of your application and displays a dialog box such as the one shown in Figure 14.

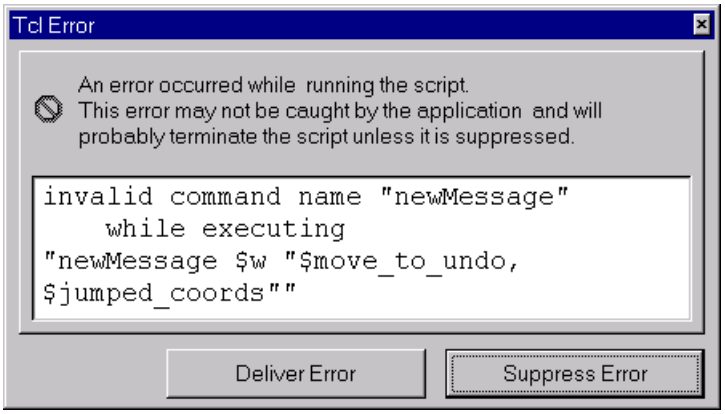


Figure 14 The TclPro Debugger Tcl Error Dialog

You have the choice of either delivering the error or suppressing the error:

Deliver Error

The application continues and the error is handled in the normal fashion for Tcl. Clicking this button is equivalent to having run the script without any debugger interference.

Suppress Error

TclPro Debugger suppresses the error, and continues executing the application. The behavior in this case is as if no error had occurred. You can continue to run or step through the application.

While your application is stopped, you can examine your Tcl script, view and change variable values, set breakpoints, and use all the other features of TclPro Debugger. If you single-step or run your application without first selecting whether to deliver or suppress the error, TclPro Debugger delivers the error if your application catches it and suppresses it otherwise.

Setting Preferences

You can specify preferences to customize TclPro Debugger. To modify Preferences, select Edit | Preferences from the menubar. Click the tabs to select your preferences for Appearance, Windows, Instrumentation, and Startup and Exit, and Browser preferences.

Appearance Preferences

The Appearance Preference tab is shown in Figure 15.

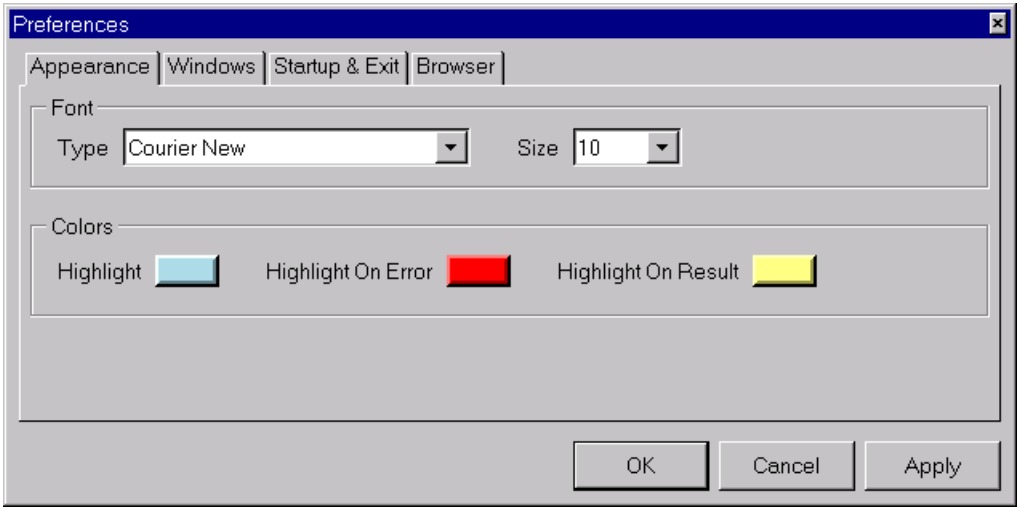


Figure 15 The Appearance Preference Tab

You can choose the following Appearance preferences:

Type — The name of the font used to display code, stack frames, variables, etc.

Size — The size of the font used to display code, stack frames, variables, etc.

Tip — TclPro Debugger attempts to optimize your font and size preferences. If you type a font that is unavailable, TclPro Debugger finds the most similar font on your computer and substitutes it. Scriptics recommends that you only use fixed-width fonts.

Note — Small font sizes can cause misalignment of the symbols in the Code Bar and their corresponding lines of code. If you experience problems, increase the font size.

Highlight — The color TclPro Debugger uses when it stops to highlight the next command it will execute

Highlight On Error — The color TclPro Debugger uses to highlight a command in which it finds an error

Highlight On Result — The color TclPro Debugger uses to highlight code it just executed after a Step To Result

After changing the Appearance tab settings, click the OK button to save your choices and close the Preferences window, the Cancel button to cancel your choices and close the Preferences window, or the Apply button to apply your choices and keep the Preferences window open.

Window Preferences

The Windows Preference tab is shown in Figure 16.

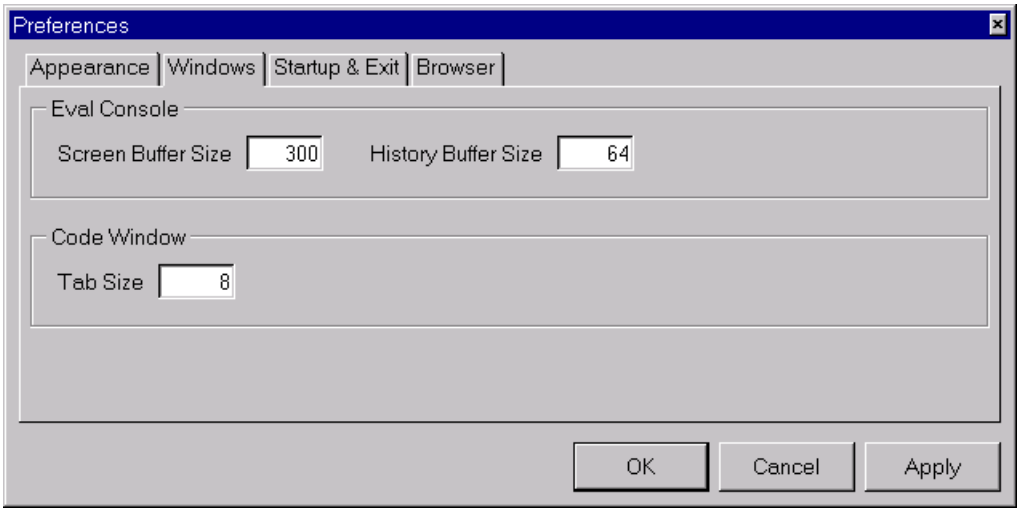


Figure 16 The Windows Preference Tab

You can modify the following Windows preferences:

Screen Buffer Size
: The number of lines of output retained by the Eval console

History Buffer Size
: The number of commands retained in the Eval console history buffer

Tab Size
: The number of characters between each tab stop.

After changing the Windows tab settings, click the OK button to save your choices and close the Preferences window, the Cancel button to cancel your choices and close the Preferences window, or the Apply button to apply your choices and keep the Preferences window open.

Startup and Exit Preferences

The Startup & Exit Preference tab is shown in Figure 17.

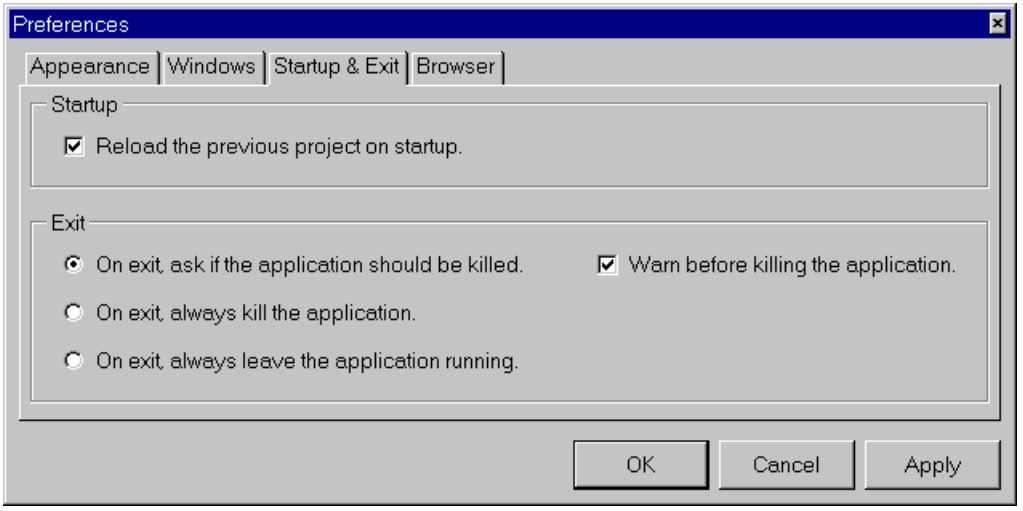


Figure 17 The Startup & Exit Preference Tab

The Startup preference controls TclPro Debugger's behavior when you start the debugger:

Reload the Previous Project on Startup
: TclPro Debugger reloads the project you had open when you last exited TclPro Debugger

The Exit preferences control TclPro Debugger's behavior when you quit the debugger:

On exit, ask if the application should be killed
: TclPro Debugger prompts you to kill the application when you exit the debugger

On exit, always kill the application
: TclPro Debugger always kills the application when you exit the debugger

On exit, always leave the application running
: TclPro Debugger leaves the application running when you exit the debugger

Warn Before Killing the Application

TclPro Debugger always prompts you when you are about to perform an action that would kill the application

After changing the Startup & Exit tab settings, click the OK button to save your choices and close the Preferences window, the Cancel button to cancel your choices and close the Preferences window, or the Apply button to apply your choices and keep the Preferences window open.

Browser Preferences

The Browser Preference tab is shown in Figure 18.

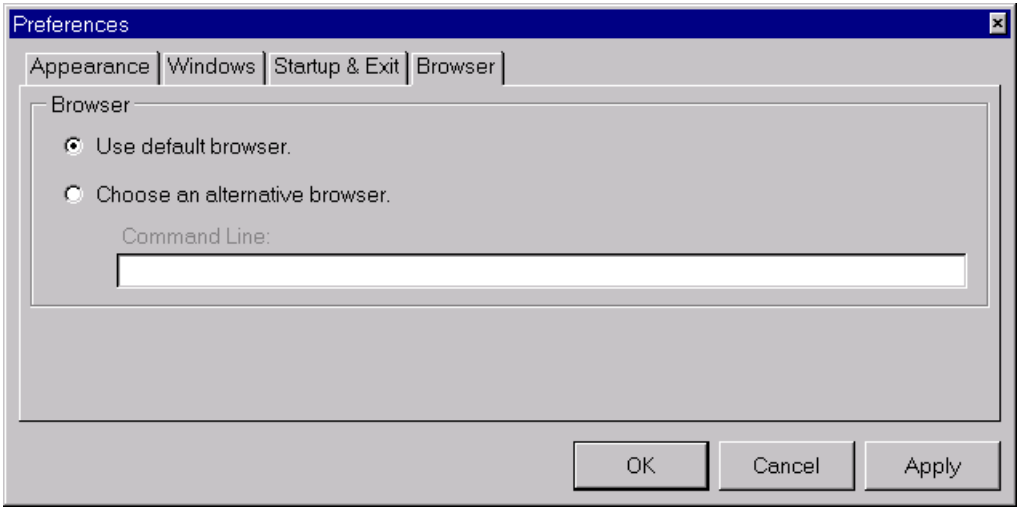


Figure 18 The Browser Preference Tab

TclPro Debugger uses a Web browser to display the Scriptics Web site when you click on the Scriptics URL in the About TclPro Debugger window and to display online help on Unix systems.

You can select one of the following choices for your Web browser with TclPro:

Use the default browser

TclPro Debugger uses the default browser on your system. (This option isn't available on Unix systems.)

Choose an alternate browser

On Windows, you see the pathname of your default browser. You can enter the pathname of an alternate browser here.

On Unix, TclPro Debugger displays a drop-down list with default command line arguments for launching Netscape and Internet Explorer. For these options to work, the executable for the browser you select must appear in one of the directories in your PATH environment variable. Alternatively, you can enter the pathname of a browser, including any flags necessary so that TclPro Debugger can launch the browser with a given URL (for example, **/usr/local/bin/netscape -no-about-splash**)

After changing the Browser tab settings, click the OK button to save your choices and close the Preferences window, the Cancel button to cancel your choices and close the Preferences window, or the Apply button to apply your choices and keep the Preferences window open.

About TclPro Instrumentation

When you begin running an application, TclPro Debugger transparently processes the specified Tcl/Tk script. It modifies the code to enable communication between TclPro Debugger and the script. This process is known as *instrumentation*. TclPro Debugger launches the application with the instrumented script in place of the original script. Scriptics designed the instrumentation to be as unobtrusive as possible. However, you can expect some slowdown in applications as a result of the instrumentation.

You can specify which procedures to instrument in the Procedures window; see "Finding Procedures" on page 38. You can also specify files and classes of procedures to leave uninstrumented; see "Changing Project Instrumentation Settings" on page 23. In addition to the files and procedures that you tell TclPro Debugger not to instrument, there are also some instances of dynamically created code that TclPro Debugger cannot instrument. These include **if** statements with computed bodies and callbacks from Tcl commands. When the application is executing uninstrumented code, it cannot communicate with TclPro Debugger. If you want to interrupt or to add a breakpoint to the script while uninstrumented code is executing, the application cannot respond until it reaches the next instrumented statement.

TclPro Debugger indicates that a procedure or file is uninstrumented by listing the procedure or file name preceded by an asterisk ("*") in the Procedures window, Windows menu, and the Code display status bar.

Debugging Remote, Embedded, and CGI Applications

In some cases, TclPro Debugger can't directly launch your application. Some examples where this is often true include CGI applications, embedded applications, and applications that must run on a system other than your debugging system.

For applications such as these, TclPro Debugger supports *remote debugging*. In remote debugging sessions, your application starts as it normally would and then establishes a special connection to TclPro Debugger. You can then use TclPro Debugger to perform all debugging tasks as you would in a local debugging session.

To debug a remote application, you must perform the following steps:

- Modify your Tcl script to work with TclPro Debugger.
- Create a remote debugging project in TclPro Debugger.
- Launch your application as you normally would. Your application establishes a connection to TclPro Debugger and you can begin your debugging session.

The following sections describe how to perform these tasks.

Modifying a Tcl Script for Remote Debugging

For your application to establish and maintain communication with TclPro Debugger, you must modify your application to **source** the *prodebug.tcl* file, which is contained in the platform-specific *bin* subdirectory of your TclPro installation (for example, *C:\Program Files\TclPro1.3\win32-ix86\bin\prodebug.tcl*). Then, your script must call the **debugger_init** procedure and, optionally, the **debugger_eval** and **debugger_break** procedures. You can modify your script in one of two ways: create a new "wrapper" script that sources your existing script, or modify your existing script.

Remote Debugging Procedures

The **debugger_init** procedure makes the initial connection with TclPro Debugger:

```
debugger_init ?host? ?port?
```

The *host* is the name of the machine on which TclPro Debugger is running. The host defaults to "localhost." The *port* is the TCP port that TclPro Debugger uses to communicate with the application. The port defaults to 2576. The **debugger_init** procedure contacts the debugger instance running on the specified host via the specified port. The **debugger_init** procedure also automatically instruments any Tcl scripts sourced by the script.

The **debugger_init** procedure returns 1 if it successfully connects to TclPro Debugger; otherwise it returns 0. You must call **debugger_init** before calling **debugger_eval** or **debugger_break**.

Note If your embedded application uses multiple subsequent interpreters, that is, it quits and restarts a Tcl interpreter more than once, each main Tcl script is treated as an individual application and must make a new connection with TclPro Debugger.

The **debugger_eval** procedure instruments Tcl code so TclPro Debugger can control the application while *script* is evaluated:

```
debugger_eval ?-name name? ?--? script
```

You can wrap your whole script inside the **debugger_eval** block. Any scripts that you **source** within a **debugger_eval** block are also instrumented.

Note The **debugger_eval** procedure behaves like the **eval** command if your application is not currently connected to TclPro Debugger.

The optional **debugger_eval -name** switch associates the tag *name* with the script. This causes TclPro Debugger to store breakpoint information as if the script were sourced from a file named *name*. This is useful when debugging remote applications or when evaluating blocks of dynamically-generated code that are used multiple times. By creating a unique name for each block, you can set breakpoints in the block that persist across invocations.

The optional **--** switch marks the end of switches. The argument following this one is treated as a script even if it starts with a "-".

The **debugger_break** procedure causes your remote application to break in much the same way as if it had encountered a breakpoint:

```
debugger_break ?message?
```

The **debugger_break** procedure is useful for debugging dynamic code. The only difference between the behavior of **debugger_break** and a line breakpoint is that **debugger_break** evaluates the *message* argument, if it is present, before breaking. When your script encounters a **debugger_break** procedure, TclPro Debugger displays a dialog box. If the *message* argument is present and not empty, TclPro Debugger displays the message string in the dialog box.

Note The **debugger_break** procedure has no effect if your application is not currently connected to TclPro Debugger.

Creating a "Wrapper" Script for Remote Debugging

If you decide to create a new script, that script should **source** the *prodebug.tcl* file, call **debugger_init**, and then **source** the file that was originally the main script of your application. This new script becomes the main script of your application. Your new main script may look like the following:

```
# Set TclProDirectory to the platform-specific bin
#   subdirectory of your TclPro distribution.

set TclProDirectory "/usr/local/TclPro1.3/solaris-sparc/bin"
source [file join $TclProDirectory prodebug.tcl]

# Assume the variables $host and $port respectively contain
#   the name of the machine on which TclPro Debugger is
#   running and the port on which it is listening.

debugger_init $host $port

# Assume $myOriginalMainScript contains the path of your
#   original script.

source $myOriginalMainScript
```

Modifying an Existing Script for Remote Debugging

If you decide to modify your existing script, you must change it to **source** the *prodebug.tcl* file and call the **debugger_init** procedure. Once **debugger_init** is called, other files sourced by the script will automatically be instrumented. If you want TclPro Debugger to instrument code in the file that calls **debugger_init**, the code that you wish to instrument must be encapsulated in a call to the **debugger_eval** procedure. See "About TclPro Instrumentation" on page 50 for more details on instrumentation.

Your new main script may look like the following:

```
# Set TclProDirectory to the platform-specific bin
#   subdirectory of your TclPro distribution

set TclProDirectory "/usr/local/TclPro1.3/solaris-sparc/bin"
source [file join $TclProDirectory prodebug.tcl]

# Assume the variables $host and $port respectively contain
#   the name of the machine on which TclPro Debugger is
#   running and the port on which it is listening.

debugger_init $host $port
debugger_eval {
# ... your code goes here ...
}
```

Creating a Remote Debugging Project

Before you begin debugging a remote application, you must create a remote debugging project in TclPro Debugger. This causes TclPro Debugger to listen on a specified port for your application to establish a connection.

To create a remote debugging project:

1) Create a new project as described in “Creating a New Project” on page 17.
2) Select the Remote Debugging option of the Project Application Settings Tab. See “Changing Project Application Settings” on page 21.
3) Enter the port number you specified in the **debugger_init** procedure in the Listen For Remote Connection On Port Number field. The default port is 2576.

Launching your Remote Application

After you have modified your application for remote debugging and created a remote debugging project in TclPro Debugger, you can launch your remote application for debugging.

Simply run your application as you would normally. Your application stops just before it evaluates the first command in the **debugger_eval** script, or the first time it sources a file, whichever comes first. TclPro Debugger displays your script in its Main window, and you can begin debugging as you would a local application.

Viewing Connection Status

You can view the connection status while debugging by selecting View | Connection Status from the menubar. TclPro Debugger displays the Connection Status window shown in Figure 19.

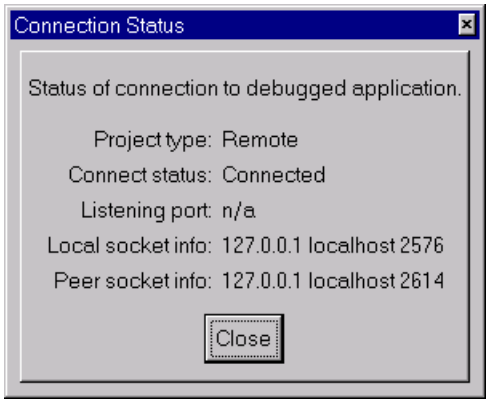


Figure 19 The Connection Status Window

The Connection Status Window displays the following information:

Project Type
: Whether the project is local or remote.

Connection Status
: Whether or not the application has established a connection to TclPro Debugger.

Listening Port
: The port number on which TclPro Debugger listens for a connection from a remote application. You can set this port for remote debugging in the Listen For Remote Connection On Port Number field of the Project Application Settings Tab. See “Changing Project Application Settings” on page 21. The default port is 2576.

Local Socket Info
: The IP address and socket number on the system running TclPro Debugger used for communication with a remote application. This is created only after a connection is established.

Peer Socket Info
: The IP address and socket number on the system running the remote application used for communication with TclPro Debugger. This is created only after a connection is established.

Using Custom Tcl Interpreters with TclPro Debugger

TclPro Debugger works properly with most custom Tcl interpreters. However, to properly instrument and execute your application, TclPro Debugger must be able to pass debugging information to your Tcl script as command-line arguments. Therefore, if your interpreter doesn't accept as its first command-line argument a Tcl script to execute or if it doesn't pass subsequent command-line arguments to the script using the standard *argc* and *argv* Tcl variables, then you must take special steps to use your interpreter with TclPro Debugger.

First, you must create a special Tcl wrapper script. The listing below shows a sample implementation of such a script for Unix systems. To use it, you must either change the line setting the *cmdPrefix* variable, replacing "tclsh" with whatever command you need to run your Tcl interpreter, or you must set your *PRODEBUG_TCLSH* environment variable to contain that command.

```
#!/bin/sh
#\
exec protclsh82 $0 ${1+"$@"}

if {$argc < 1} {
    puts stderr "wrong # args: location of appLaunch.tcl is required"
}

if {[info exists env(PRODEBUG_TCLSH)]} {
    set cmdPrefix "$env(PRODEBUG_TCLSH)"
} else {
    set cmdPrefix "tclsh"
}

set customScriptName "/tmp/launchScript.[pid]"
set appLaunchPath [lindex $argv 0]

set f [open $customScriptName w]
puts $f "
file delete -force $customScriptName
set argv0 [list $appLaunchPath]
set argv [list [lrange $argv 1 end]]
set argc \[llength \$argv\]
source \$argv0
"
close $f

catch {
    eval exec $cmdPrefix [list $customScriptName 2>@stderr >@stdout <@stdin]
}
```

Then, to debug your application select the wrapper script as your interpreter (that is, type the path and name of the wrapper script in the Interpreter field of the Project Application Settings Tab). Specify the script and any script arguments for your application in the Project Application Settings Tab as normal.

Chapter 4
TclPro Checker

TclPro Checker helps you find errors in a Tcl script quickly before you run the script. Using TclPro Checker can help you find problems in new scripts, in scripts from older versions of Tcl/Tk, or in scripts that you have ported from another operating system. You can use TclPro Checker to assess the quality of a body of Tcl code or to quickly examine large Tcl files. TclPro Checker also warns about potential incompatibilities to help you upgrade applications to the latest releases of Tcl, Tk, and [incr Tcl].

Supported Tcl Versions

By default, TclPro Checker verifies scripts written for Tcl version 8.2. You can use TclPro Checker with the packages and versions of Tcl, Tk, and [incr Tcl] listed in Table 2.

Table 2 Packages and Version Numbers

Tcl	Tk	[incr Tcl]	Expect	TclX
7.3	3.6	1.5	n/a	n/a
7.4	4.0	2.0	n/a	n/a
7.5	4.1	2.1	n/a	n/a
7.6	4.2	2.2	na/	na/
8.0	8.0	3.0	5.28	8.0
8.1	8.1	n/a	5.29 or 5.30	8.1
8.2 (default)	8.2	3.1	5.31	8.2

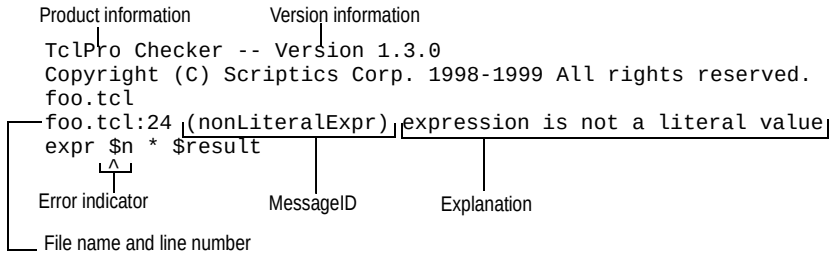
Note Expect command names that "collide" with command names in the Tcl/Tk core (for example, **send**) can confuse TclPro Checker when it checks an Expect script, causing it to report syntax errors. To avoid this, use the **exp_** prefix for all such ambiguous commands (for example, use **exp_send** instead of **send**).

Using TclPro Checker

To check a file using TclPro Checker, type the **procheck** command with a file name, for example:

```
procheck foo.tcl
```

If your code contains errors or warnings, TclPro Checker provides feedback by default that looks similar to this:



You can specify multiple file names on the same line, for example:

```
procheck foo1.tcl foo2.tcl
```

To check all the files in a directory, use the asterisk ("*") with the *.tcl* file extension, for example:

```
procheck *.tcl
```

If you don't specify any files, **procheck** expects input from standard input.

For other examples of output, see "Examples of Output from TclPro Checker" on page 65.

One-Pass and Two-Pass Checking

By default, TclPro Checker performs a two-pass scan of your scripts. The first pass accumulates information about user-defined procedures and user-defined [incr Tcl] classes. This information includes:

- the number of arguments for procedure definitions and [incr Tcl] constructor definitions
- the scope of procedures (namespace, protection level)

- redefinition of procedures using the Tcl **rename** command
- imports and exports of namespace procedures
- class structures of inherited [incr Tcl] classes

The second pass uses this information to provide warnings and error messages concerning the usage of the user-defined procedures, including:

- calling a procedure with the wrong number of arguments
- calling an [incr Tcl] class constructor with the wrong number of arguments
- redefining existing procedures, by either the **rename** command or by defining a procedure or class with and identical name
- calling [incr Tcl] class procedures out of scope
- calling class procedures with invalid permissions (private or protected)

TclPro Checker properly handles all variations of user-defined procedures in namespaces.

Note TclPro Checker does not currently check the following:

- variable usage (for example, attempting to use the value of an undefined variable or attempting to perform math operations on a list variable)
- [incr Tcl] class methods
- argument types passed to user-defined procedures
- redefinition of built-in Tcl, Tk, or [incr Tcl] commands

Also, if you define a procedure multiple times, TclPro Checker generates a usage error when calling that procedure only if the call fails to match *any* of procedure definitions. Because of the dynamic nature of procedure definition and redefinition, TclPro Checker can't determine which argument list is currently valid for the given procedure call.

TclPro Checker does not automatically scan scripts that are sourced by your script. Therefore, you must include on the **procheck** command line all files that define user procedures and classes used by your script.

For a quicker but less comprehensive check of your scripts, you can use the **procheck -onepass** option to force TclPro Checker to perform a one-pass scan of your scripts. A one-pass scan does not check for any of the potential errors or misuses of user-defined procedures and [incr Tcl] classes described above.

You can also use the **procheck -verbose** option to get a list of all commands used by the scripts you specify that are not defined in that collection of scripts. If you don't include the **-verbose** option, TclPro Checker doesn't warn you about undefined procedures.

TclPro Checker Messages

TclPro Checker examines your code and displays a message describing each error or potential mistake that it detects. Depending on the type of script that you are checking, you may want to limit the types of problems that it reports rather than see the entire output from TclPro Checker.

Each message generated by TclPro Checker lists the file and the line number where the error or warning occurred, a *messageID*, a description of the error or warning, and an error indicator, which is a caret ("^") that indicates the code fragment in which the error occurred. The messageID is the word in parentheses just after the file and line number information. It provides information about the type of problem that generated the error or warning, which are listed below. Using TclPro Checker you can specify types of errors and warning that you want to suppress, which allows you to focus to more strategic errors or warnings. TclPro Checker provides suggestions, when possible, on ways to fix the problems that it indicates in the error or warning text. In the example in Figure 20, Checker indicates that there is a missing close brace.

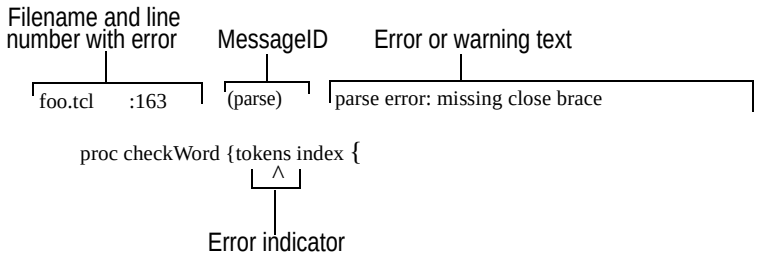


Figure 20 Anatomy of a TclPro Checker Message

You can limit output in the following ways:

- Limiting errors and warnings by type: an error is either a parsing or syntax error, and warnings indicate possible problems with platform portability, upgrade, performance, or usage issues. Warnings indicate code fragments where there may be an error, but the code fragment may be appropriate in its context.
- Specifying groups of messages to suppress, for example, you might want to suppress messages related to usage warnings.

Controlling Feedback on Errors and Warnings

Messages are grouped into two types of errors and four types of warnings.

Parsing Errors

TclPro Checker generates a parsing error when it encounters commands that cannot be parsed by the Tcl parser, such as a missing curly brace or badly formed list. For example: the following code generates a parsing error because it is missing a quote at the end of the **puts** statement:

```
proc foo {} {
    puts "hello
}
```

In cases like this, TclPro Checker attempts to move past the procedure where the parsing error was found, and continue to check additional commands after the parsing error.

Syntax Errors

TclPro Checker generates a syntax error when it encounters any errors that will cause your script to fail, such as the wrong number of arguments or invalid types or options. For example, the following code generates a syntax error is because the wrong number of arguments are supplied:

```
set x 3 45
```

Only commands defined in Tcl, Tk, or [incr Tcl] are checked for syntax errors.

Platform Portability Warnings

TclPro Checker generates warnings when a command is used that may be nonportable between various platforms.

```
set file [open $dir/$file r]
```

In this example, the **file join** command should be used so that the correct directory and file separator is used, that is, “\” on Windows and “/” on Unix.

Suggestion for Upgrading

Upgrade warnings indicate features that have changed in a later version.

```
namespace foo {
    variable bar 0
}
```

When [incr Tcl] was upgraded to 3.0, it inherited the Tcl namespace command. The syntax of defining a namespace has changed from older versions of [incr Tcl] because of this. With earlier versions of [incr Tcl], correct usage was:

```
namespace foo {body}
```

With [incr Tcl] 3.0 and later, correct usage is shown below:

```
namespace eval foo {body}
```

Performance Warnings

TclPro Checker generates a warning when a performance-optimization opportunity is detected. For example: if your code included:

```
set x [expr $x * $y]
```

it would generate a performance warning because performance is improved with curly braces, as shown below:

```
set x [expr {$x * $y}]
```

Usage Warnings

TclPro Checker generates a warning when a command is used in a manner that is possibly incorrect but is still syntactically legal. For example, the **incr** command expects a value and not a reference below:

```
incr $counter
```

Warning and Error Flags

You can control which types of errors and warnings are listed by TclPro Checker by specifying one of the **-W** flags on the TclPro Checker command line. Table 3 shows the flags that control the level of messages for warning and errors.

Table 3 TclPro Checker Warning and Error Flags

Flag	Description
-W1	Display parsing and syntax errors.
-W2	Display parsing and syntax errors, and usage warnings.
-W3	Display parsing and syntax errors, portability warnings, upgrade warnings, performance warnings, and usage warnings.
-Wall	Displays all messages and errors. This is the default.

As an example, the first time you check your script you might want to display only errors but not warnings. You might first run TclPro Checker with the **-W1** flag, which only displays parsing and syntax errors, but does not display any warnings. After examining the output from running with the **-W1** flag and fixing any errors that were reported, you might run with the **-W2** flag to see a variety of additional warnings.

Suppressing Specific Messages

Each warning or error message has an associated messageID. You can filter out the display specific warnings or errors by specifying **-suppress** to prevent that type of messageID from being displayed. You might want to filter out certain messages because they point out items that do not apply to the script that you are checking, for example: if you are porting a script to only one platform, you do not care whether your script has portability issues.

In the following example, the messageID is “nonPortCmd”:

```
foo:tcl:53 (nonPortCmd) use of non-portable command
registry values $key
^
```

You can suppress this type of message by specifying **-suppress nonPortCmd** on the command line, for example:

```
procheck -suppress nonPortcmd foo.tcl
```

Tip You can suppress multiple messageID types at the same time by specifying **-suppress** with the multiple instances of messageIDs in quotation marks, for example:

```
procheck -suppress "nonLiteralExpr badOption" foo.tcl
```

You can also specify **-suppress** with the messageID for each instance of the message ID that you want to filter, for example:

```
procheck -suppress nonLiteralExpr -suppress badOption foo.tcl
```

For a complete list of all the messageIDs, see Appendix B, “TclPro Checker Messages.”

Examples of Output from TclPro Checker

To provide examples of TclPro Checker output, here is the sample script, *foo.tcl*, that is checked in the examples that follow:

```
set $y 3
set x [expr $y + 5]
set x y z

if {$x > 6}
{
    puts out "world"
}

proc foo {args bar} {
    puts "hello, world"
}

proc p {{a 0} b} {
    puts -nonew "hello"
}
```

Specifying Verbose Feedback

You can specify the **-verbose** argument when you run TclPro Checker. When you specify **-verbose**, TclPro Checker displays the error information in three lines and the version and summary information when TclPro Checker exits, for example:

```
procheck -verbose foo.tcl
```

The feedback from the command line with **-verbose** specified looks similar to this:

```
TclPro Checker -- Version 1.3.0
Copyright (C) Scriptics Corp. 1998-1999. All rights reserved.
This product is registered to: Sinking Ships, Ltd.

scanning: /home/kenj/test/foo.tcl
checking: /home/kenj/test/foo.tcl
foo.tcl:1 (warnVarRef) variable reference used where variable name expected
set $y 3
    ^
foo.tcl:2 (warnExpr) use curly braces to avoid double substitution
expr $y + 5
     ^
foo.tcl:3 (numArgs) wrong # args
set x y z
^
foo.tcl:5 (noScript) missing a script after "if"
if {$x > 6}
           ^
foo.tcl:6 (warnUndefProc) undefined procedure:
    puts out "world"

{
^
```

```
foo.tcl:10 (argAfterArgs) argument specified after "args"
proc foo {args bar} {
^
foo.tcl:14 (nonDefAfterDef) non-default arg specified after default
proc p {{a 0} b} {
       ^

Packages Checked | Version
-----------------|--------
tcl                8.2
tk                 8.2
expect             5.31
[incr Tcl]         3.1
tclX               8.2

Number of Errors:   4
Number of Warnings: 3

Commands that were called but never defined:
--------------------------------------------

    puts out "world"
```

Specifying Quiet Feedback

You can specify the **-quiet** argument when you run TclPro Checker. When you specify **-quiet**, TclPro Checker displays the basic error information on one line with the messageID, instead of the three-line output that includes the code body and the error indicator, for example:

```
procheck -quiet foo.tcl
```

The output with the **-quiet** argument appears as follows:

```
TclPro Checker -- Version 1.3.0
Copyright (C) Scriptics Corp. 1998-1999. All rights reserved.
This product is registered to: Sinking Ships, Ltd.

foo.tcl:1 (warnVarRef) variable reference used where variable name expected
foo.tcl:2 (warnExpr) use curly braces to avoid double substitution
foo.tcl:3 (numArgs) wrong # args
foo.tcl:5 (noScript) missing a script after "if"
foo.tcl:6 (warnUndefProc) undefined procedure:
    puts out "world"

foo.tcl:10 (argAfterArgs) argument specified after "args"
foo.tcl:14 (nonDefAfterDef) non-default arg specified after default
```

Specifying Use of Older Versions

You can run TclPro Checker and specify **-use** with an older version of Tcl or Tk. To check for older versions of any package, use the **-use** option and specify the version to check. For example, to check a file written for Tcl7.5 and Tk4.1, enter:

```
procheck -use "tcl7.5" -use "tk4.1" foo.tcl
```

Valid **-use** arguments are package names followed by a version number. Supported package names are “tcl”, “tk”, “expect”, “incrTcl”, and “tclX”. Table 2, “Packages and Version Numbers” on page 59, lists the versions supported for each package. If you do not specify a version for a package, TclPro Checker uses the version compatible with the Tcl version you select.

Note Tk, [incr Tcl], TclX, and Expect are checked *only* if you explicitly specify them on the command line with **-use** option.

When you specify older versions of Tcl and any extension (including Tk), the versions of Tcl and any specified extension must be compatible, as listed in Table 2. The following example includes incompatible versions and should not be used:

```
procheck -use "tcl7.5" -use "tk4.0" foo.tcl
```

The correct version pair is:

```
procheck -use "tcl7.5" -use "tk4.1" foo.tcl
```

Error Checking

The command line in following example requests **-W1** error checking, which includes only parsing and syntax errors:

```
procheck -W1 foo.tcl
```

The feedback from the command line with **-W1** specified looks similar to this:

```
TclPro Checker -- Version 1.3.0
Copyright (C) Scriptics Corp. 1998-1999. All rights reserved.


foo.tcl
foo.tcl:5 (numArgs) wrong # args
set x y z
^
foo.tcl:7 (noScript) missing a script
if {$x > 6}
^
```

Error and Warning Checking

The command line in following example requests **-W2** error checking, which includes parsing errors, syntax errors, upgrade warnings, and performance warnings.

```
procheck -W2 foo.tcl
```

The feedback from the command line with **-W2** specified looks similar to this:

```
TclPro Checker -- Version 1.3.0
Copyright (C) Scriptics Corp. 1998-1999. All rights reserved.
This product is registered to: Sinking Ships, Ltd.

scanning: /home/kenj/test/foo.tcl
checking: /home/kenj/test/foo.tcl
foo.tcl:3 (numArgs) wrong # args
set x y z
^
foo.tcl:5 (noScript) missing a script after "if"
if {$x > 6}
           ^
foo.tcl:10 (argAfterArgs) argument specified after "args"
proc foo {args bar} {
         ^
foo.tcl:14 (nonDefAfterDef) non-default arg specified after default
proc p {{a 0} b} {
       ^
```

Checking for All Warnings and Errors

The command line in following example requests **-W3** error checking, which includes parsing errors, syntax errors, upgrade, portability, and performance warnings.

```
procheck -W3 foo.tcl
```

The feedback from the command line with **-W3** specified looks similar to this:

```
TclPro Checker -- Version 1.3.0
Copyright (C) Scriptics Corp. 1998-1999. All rights reserved.
This product is registered to: Sinking Ships, Ltd.

scanning: /home/kenj/test/foo.tcl
checking: /home/kenj/test/foo.tcl
foo.tcl:1 (warnVarRef) variable reference used where variable name expected
set $y 3
    ^
foo.tcl:2 (warnExpr) use curly braces to avoid double substitution
expr $y + 5
     ^
foo.tcl:3 (numArgs) wrong # args
```

```
set x y z
^
foo.tcl:5 (noScript) missing a script after "if"
if {$x > 6}
           ^
foo.tcl:6 (warnUndefProc) undefined procedure:
    puts out "world"

{
^
foo.tcl:10 (argAfterArgs) argument specified after "args"
proc foo {args bar} {
         ^
foo.tcl:14 (nonDefAfterDef) non-default arg specified after default
proc p {{a 0} b} {
       ^
```

Chapter 5
TclPro Compiler

Traditionally Tcl code has been distributed in source form. This had the advantage of being simple to use and allowing users to customize the code, but it had some disadvantages: you can't keep proprietary information secret and it may be harder to support users if they modify the code. TclPro Compiler eliminates these disadvantages by translating the Tcl scripts into bytecode format. You can distribute bytecode files to users to protect your intellectual property and simplify support.

Supported Versions

You must use Tcl/Tk 8.2 to load programs compiled with TclPro Compiler 1.3.

TclPro Compiler 1.3 generates bytecode files in version 1.3 format (to support Tcl 8.2). These new bytecode files require version 1.3 of the **tbcload** package. The **tbcload** 1.3 package supports the following bytecode file formats:

- 1.3 (generated by TclPro Compiler 1.3)
- 1.0.1 (generated by TclPro Compiler 1.2)

The **tbcload** 1.3 package does *not* support version 1.0 bytecode files (generated by TclPro Compiler 1.0). See "Distributing Bytecode Files" on page 78 for more information on the **tbcload** package.

Overview

Tcl code was traditionally interpreted on an as-needed basis. Before Tcl Version 8.0, the Tcl core did not include an internal compiler. Tcl Version 8.0 included a compiler; however, this compiler was internal to the interpreter, and compiled scripts could not be saved for later use. TclPro Compiler lets you compile scripts independently of execution, then store them so you can load and execute the bytecode file when you want to.

When you use TclPro Compiler, the bytecode file is stored as Tcl byte codes with the default extension *.tbc*. For example: if you compile the script *foo.tcl* with TclPro Compiler, the bytecode file is stored as *foo.tbc*. When you want to use the bytecode file, you can **source** it without spending the time to recompile *foo.tcl*.

You can distribute a bytecode file; this allows you to avoid shipping the Tcl source code, thus keeping your code secure. Bytecode files can also be used with TclPro Wrapper to create bundled applications that don't require special installation; see Chapter 6, "TclPro Wrapper."

Compiling your code

TclPro Compiler compiles Tcl files, and after compiling, creates an output file with the *.tbc* extension. To compile a Tcl script, enter:

```
C:> procomp filename.tcl
```

This command creates the output file *filename.tbc*.

You can specify multiple file names on the command line; the bytecode files will have the same names as the input file with extension *.tbc*. You can also use wildcard specifications in the file names following the **glob** conventions. For example: to compile all *.tcl* files in *C:\dir1*, type:

```
C:> procomp c:\dir1\*.tcl
```

When a file is compiled, the output file is placed in the same directory as the input file, with the same name, and extension *.tbc*.

To rename a file while compiling it, use the **-out** flag to create a single file with a custom name. You specify the command in the form: **procomp -out** *newfilename oldfilename*, for example: to rename *foo.tcl* to *bar.tst*, you would type:

```
C:> procomp -out bar.tst foo.tcl
```

The **-out** flag can also specify a directory, for example: the following command:

```
C:> procomp -out c:\dir2 c:\dir1\*.tcl
```

generates the set of files with the same name with the *.tbc* extension, but the files are placed in *C\:dir2*.

Note You can only specify a single input if the **-out** flag does not specify a directory.

You can also force TclPro Compiler to overwrite all output files that already exist using the **-force** flag. This flag deletes the output file before running TclPro Compiler to ensure that the compilation does not fail because of permission errors. Because TclPro Compiler creates the output file with the same permissions as the input file, the *.tbc* file generated from a read-only *.tcl* file is also read-only. As a result, recompiling a read-only file will fail unless you specify the **-force** flag.

Bytecode Files

TclPro Compiler creates an internal representation of the Tcl script using the Tcl bytecode compiler that is built into the Tcl core. It performs additional computations, and then emits a representation of the bytecode file to the output file. The output file itself is a simple Tcl script that loads the bytecode run-time package, **tbcload**, and then invokes a command in that package to load and run the bytecode files.

Bytecode files are just Tcl scripts. This allows you to use bytecodes anywhere you would use Tcl scripts. For example: you can **source** bytecode files. You can store a *.tbc* script in a Tcl variable, for example, by reading the *.tbc* file or reading it from a socket and then execute it using the **eval** command. You can use the *.tbc* scripts to drive **protclsh82** or **prowish82**.

Prepending Prefix Text

Because the bytecode file is a Tcl script, there might be situations where you might want to add some specialized setup code at the start of the script. For example, if you want to directly execute a script file under Unix it should start with the following lines:

```
#!/bin/sh
# the next line restarts using protclsh82 \
exec protclsh82 "$0" "$@"
```

See the manual page for **protclsh82** for more information. By default, TclPro Compiler preserves everything from the start of the file to the first non-blank or non-comment line. Therefore in this example, TclPro Compiler adds these three lines to the top of the script it generates.

You can override this default behavior with the **-prefix** option. controls which prefix string is prepended to the output file. Table 4 lists the **-prefix** options available.

Table 4 TclPro Compiler **-prefix** options

Type	Function
none	Do not add a prefix string.
auto	Extract the prefix from the input file; everything from the start of the file to the first non-comment line is prepended to the output file. (**Default**)
tag	Extract the prefix from the input file; everything from the start of the file to the first occurrence of a comment line starting with the text "Tcl::Compiler::Include" is prepended to the output file.
filename	Extract the prefix text from a specified file.

See the *procomp.1* manual page for more information.

Changes in Behavior

There are few differences between the behavior of bytecode files and Tcl scripts that are not compiled. This section explains these differences.

TclPro Compiler has the following limitations:

- Only those procedures that are defined at the top level can be compiled.
- The **info body** command on compiled procedures does not provide meaningful information; see "Example 1: Cloning Procedures" on page 75

However, these limitation do not prevent the affected procedures from being compiled at runtime. The contents of the bytecode file are a representation of the internal structures of the compiled Tcl script, without the source code. Procedures defined in the source file are compiled and their internal structures are also stored without source code. Thus, compiled procedure bodies cannot be read or accessed through the **info body** command. As a consequence, you cannot depend on being able to read procedure bodies in the bytecode, as shown in Example 1.

The command **info body** on a compiled procedure cannot return the actual body of the procedure because that information is not available. Instead, it returns a fabricated script containing:

- A comment, which identifies this body of code as belonging to a compiled procedure.
- An error command: this is used as an aide in detecting unsupported uses of **info body**, as shown in Example 1.

Example 1: Cloning Procedures

Scripts that use the bodies of procedures in computations will not work properly if the procedures have been compiled. For example, the script below uses **info body** to extract the body of one procedure and use it to create another procedure that is identical.

```
#clone.tcl--
proc len {a} {
    return [string length $a]
}
proc len1 {a} [info body len]
puts "[len {abc}] + [len1 {monkey}]"
```

The two calls to **proc** create two procedures, **len** and **len1**, with identical bodies.

If you run the *clone.tcl* file, you get this output:

```
C:> protclsh82 clone.tcl
3 + 6
```

Bytecode files, however, do not contain any sources for compiled procedure bodies, and **info body** returns a standard value.

If you run the *clone.tbc* file, you get this output:

```
C:> protclsh82 clone.tbc
called a copy of a compiled script
while executing
"error "called a copy of a compiled script""
(procedure "len1" line 2)
invoked from within
"# Compiled -- no source code available
error "called a copy of a compiled script""
invoked from within
"tbcload::bceval {
TclPro ByteCode 1 0 1.3 8.2
6 0 49 12 0 0 28 0 6 6 6 -1 -1
49
/QE<!(H&s!/HW<!'E'<!*Ki<!/'vpv1fAs!+EE<!2o8X!0fA9v4u8X!1'8X!z
6=t-Ow+..."
(file "clone.tbc" line 17)
```

Note that the call to **len1** resulted in an error being thrown; this error comes from the script returned by the **info body len** command. The script throws the error rather than failing silently to help you to detect unsupported uses of **info body** command. If you need to use the body of a procedure in a computation, do not compile that procedure.

What is and isn't Compiled

TclPro Compiler will compile most of the Tcl code in your applications, but it can't compile absolutely every Tcl command. Where TclPro Compiler cannot compile a command it leaves it in text form where it will be compiled at runtime when the command is invoked. Your bytecode files will still execute correctly even if some commands aren't compiled, but uncompiled commands mean that part of your source is more easily accessible to your users. This section discusses what TclPro Compiler can and cannot compile.

When it compiles a script, TclPro Compiler divides the script up into its component Tcl commands and compiles each one. If TclPro Compiler can determine that the argument to a command is a Tcl script, then it compiles that script also. However, if TclPro Compiler can't determine that an argument is a script, then it leaves that argument as a string. For example, TclPro Compiler can identify all the Tcl scripts used as arguments to standard Tcl commands, such as the bodies of **if**, **while**, and **proc** commands. However, in the following script TclPro Compiler can't tell that the argument to the **do10** procedure is a script:

```
proc do10 {script} {
    for {set i 1} {$i <= 10} {incr i} {
        eval $script
    }
}
do10 {puts "hello"}
```

In general, if you write a procedure that takes a script as an argument, TclPro Compiler can't tell that the argument is a script, rather than, say, an ordinary string value, so it can't compile that argument. Again, the bytecode file will behave correctly; the unknown argument will be compiled when it is actually executed.

TclPro Compiler has these limitations:

- [incr Tcl] code is not compiled.
- Bodies of dynamically created procedures cannot be compiled.
- Procedures within the scope of **namespace eval** are not compiled

The following example illustrates the constraints with procedures and namespaces.

Example 2: Procedures used with Namespace

TclPro Compiler does not currently understand the **namespace eval** command enough to know that arguments to **namespace eval** form a Tcl script, so that nothing that follows **namespace eval** is compiled, including procedures.

Example 2 shows two procedures: a procedure defined inside a **namespace eval** construct and one defined outside it. In this example, **namespace eval** prevents procedure bodies from being compiled.

```
# Example2.tcl--
namespace eval sample {
    namespace export not_compiled compiled

    proc not_compiled {a1 a2} {
        return [list $a1 $a2]
    }
}
    proc sample::compiled {a1 a2} {
    puts "hello"
}
```

Compiler Components

TclPro Compiler is made up of two components:

- TclPro Compiler generates a bytecode file from a Tcl script containing internal structures. See “Creating Package Indexes” on page 77.
- The runtime loader, **tbcload**, takes the bytecode file, loads the bytecodes into an interpreter, and executes them. See “Distributing Bytecode Files” on page 78.

Creating Package Indexes

After you compile Tcl package scripts into *.tbc* files, you can use the **pkg_mkIndex** command to create package index files for your *.tbc* files. After creating the index files, users of your package will transparently load your bytecode files instead of the original script. Creating package index files for *.tbc* files requires the **pkg_mkIndex -load tbcload** option:

```
C:> pkg_mkIndex -load tbcload $dir *.tbc
```

Important You must use Tcl 8.0.5 or later to create package index files for your *.tbc* files.

Distributing Bytecode Files

Compiled *.tbc* files execute a **package require tbcload** command. The **tbcload** package must be accessible via standard package loading mechanisms in order for the *.tbc* file to be interpreted successfully.

Because the **protclsh82** and **prowish82** interpreters include the **tbcload** package, **tbcload** is found automatically when the *.tbc* files are processed by these interpreters. There might be situations where you are unable to or do not want to use the **prowish82** or **protclsh82** interpreters, for example: if you are creating your own Tcl/Tk extensions, or if **prowish82** or **protclsh82** are too large to distribute to your customers.

The **tbcload** package is available as a shared library (such as a *.dll* on Windows and *.so* on Solaris) and as a static library. The shared library exports the two package initialization procedures: **Tbcload_Init** and **Tbcload_SafeInit**, which are required by the Tcl **load** command. You can use the shared library as you would any other Tcl package:

- Use **pkg_mkIndex** to create a package index file.
- Make sure that the shared library and index file are placed in a directory accessible to the package load mechanism.

If you follow the above guidelines, you can ship your bytecode files and the **tbcload** shared library to customers. See “Supported Versions” on page 71 for information on compatible versions of Tcl/Tk, **tbcload**, and the bytecode files.

If you are building you own extensions, you can either use **tbcload** as a dynamically loaded Tcl package as described above, or you can add it to your application as a static package. In the latter case, your **Tcl_AppInit** procedure must contain the following code:

```
#include <proTbcLoad.h>

if (Tbcload_Init(interp) == TCL_ERROR) {
    return TCL_ERROR;
}
Tcl_StaticPackage(interp, "tbcload", Tbcload_Init, Tbcload_SafeInit);
```

Compilation Errors

TclPro Compiler provides an added check that your code is syntactically correct. A benefit of compiling procedure bodies in advance is that some syntax errors are caught at compilation rather than at runtime. Because Tcl procedures in standard

Tcl code are compiled on an as-needed basis, errors are not caught until you run the procedures. TclPro Compiler informs you of errors that are caught when it compiles the file.

This example shows an error message from a compilation. The file contains syntactically incorrect Tcl code.

```
Sample for a bad file (fail.tcl):
# note the missing close-brace
set msg {
```

If you run this code in an interpreter, you see the following error message:

```
% protclsh82.exe fail.tcl
missing close-brace
    while compiling
"set msg { ..."
    (file "fail.tcl" line 15)
```

If you compile, you get this output:

```
compilation of "fail.tcl" failed: missing close-brace
```

TclPro Compiler saves the error generated by the compilation. In this example, TclPro Compiler displays the string “missing close-brace” and displays the error message. You will need to fix syntax errors like this one before TclPro Compiler can compile the script. For help in tracking down errors, see Chapter 4, “TclPro Checker.”

Chapter 6
TclPro Wrapper

An application that you write in Tcl can consist of many components, such as:

- One or more Tcl scripts
- Either a standard or a custom Tcl interpreter
- The standard Tcl libraries and support files (for example, *init.tcl*)
- Optionally, the standard Tk libraries and support files
- Optionally, one or more extensions implemented as libraries of Tcl scripts
- Optionally, additional data files such as bitmaps

Traditionally, if you wanted to distribute an application that you wrote in Tcl, you would need to make sure that all of the files listed above that your application used were installed on your target system. You would also need to make sure that the system was configured properly so that your application could find all of the files it needed.

TclPro Wrapper can greatly simplify the process of distributing an application that you write in Tcl. TclPro Wrapper is a tool that collects all of the files needed to run a Tcl application—such as Tcl scripts, graphics and other data files, Tcl extensions, a Tcl interpreter, and the standard Tcl and Tk libraries—into a single executable file, which is called a *wrapped application*. A user can then install this file anywhere on their system and execute it without needing to install any other packages or otherwise configure their system.

You invoke TclPro Wrapper using the **prowrap** command from the command line. For example, the following command creates an executable named *myApp.exe* that contains a **wish** interpreter, the standard Tcl and Tk libraries, the Tcl scripts *myApp.tcl* and *help.tcl*, and several GIF images from a subdirectory named *images*:

```
C:> prowrap -out myApp.exe myApp.tcl help.tcl images\*.gif
```

Executing the resulting *myApp.exe* file is equivalent to entering:

```
C:> wish myApp.tcl
```

How the Internal File Archive Works in a Wrapped Application

The internal file archive of a wrapped application contains all Tcl scripts and data files that you specify when you wrap an application. TclPro Wrapper incorporates special support into the wrapped application that allows Tcl scripts in the wrapped application to access files in the internal file archive just as if they were stored individually on disk. In other words, your Tcl scripts in a wrapped application can execute standard Tcl commands such as **source** and **open** to access files in the internal file archive.

Note The files in the internal file archive are read-only.

Whether your Tcl script attempts to access a file from the internal file archive or from disk is determined by the following rules:

- If you attempt to access a file using an absolute pathname (for example, */user/kate/images/widget.gif*), then your Tcl script *always* looks for the file on your disk.
- If you attempt to access a file using a relative pathname (for example, *images/widget2.gif*), then your Tcl script first looks for the file in the internal file archive. If it finds a file in the archive with the *exact* relative pathname specified, then it uses that file; otherwise, it looks for the file on your disk.

By default, files that you specify in your **prowrap** command with relative pathnames retain that pathname in the archive. Files that you specify with absolute pathnames are stripped of their drive and root directory characters. You can also modify this behavior by using the **prowrap -relativeto** argument. See "Determining Path References in Wrapped Applications" on page 86 for information on how pathnames for files in the internal archive of a wrapped application are determined.

Important The internal file archive isn't a full-fledged filesystem. Instead, the files are stored in the equivalent of a flat table. This has several important implications for accessing files in the archive:

- The current working directory of your Tcl script has no relevance to the pathname you should use to access a file in the archive. For example, if there is a file in the archive that you wrapped with the relative pathname *interface/main.tcl*, then the two source commands in the following code fragment both access that same file in the archive:

```
cd /tmp
source interface/main.tcl
# This accesses the same file as above in a wrapped application
cd /usr/local/bin
source interface/main.tcl
```

- The Tcl **glob** command doesn't match any files in the archive. For example, if you wrap the files *images/card1.gif* and *images/card2.gif*, the **glob** pattern "images/*.gif" fails to match either of these files. If you have an application that depends on the **glob** command to produce arbitrary lists of wrapped files, you need to rewrite it to use explicit lists of wrapped files. If you use a variable to contain the file list, one technique you can use is to set the value of the variable when you wrap the application using the **prowrap -code** option. The following example uses the Unix back-quote command evaluation and shell filename expansion techniques to set the variable *imageList* to contain a list of files in the wrapped *images* directory:

  ```
  % prowrap myApp.tcl images/*.gif \
  -code "set imageList [list 'echo images/*.gif']"
  ```

- If you attempt to access a file on disk using a relative pathname, and there happens to be a file in the archive with the same pathname, your Tcl script accesses the file in the archive rather than the file on the disk. This is referred to as *file shadowing*.
- If you attempt to access a file in the archive and a file with that pathname does not exist, then your Tcl script attempts to access the file on disk. This is referred to as *fall-through*.

"Changing File References" on page 102 provides guidelines for writing your applications so that they use wrapped files and unwrapped files properly.

Wrapping an Application

This section describes how to wrap your application.

Wrapping Tcl Scripts and Data Files

To wrap one or more Tcl scripts and any associated data files (for example, bitmaps), simply list all the files as arguments to the **prowrap** command. For example, suppose you have an application consisting of a single script file, *app.tcl*. To wrap it, enter:

```
C:> prowrap app.tcl
```

This creates a wrapped application called *prowrapout.exe* on Windows systems or *prowrapout* on Unix systems. When you run the wrapped application, it uses **wish** to execute your *app.tcl* script. In other words, running the wrapped application in this case is the same as executing:

```
C:> wish app.tcl
```

By default, **prowrap** includes in your wrapped application a customized **wish** Tcl interpreter with built-in support for the [incr Tcl], [incr Tk], [incr Widgets], TclX, and Expect (Unix systems only) extensions. “Specifying the Tcl Interpreter” on page 84 describes how you can specify a different Tcl interpreter

If your application has several script files, just include them on the **prowrap** command line. For example, if *app.tcl* sources the files *utils.tcl* and *help.tcl* from the *aux* subdirectory, you can wrap them with the following command:

```
C:> prowrap app.tcl aux\utils.tcl aux\help.tcl
```

Important By default, your wrapped application sources the first file you list in the **prowrap** command. So in this example, when you execute your wrapped application, it sources *app.tcl*. You can change this behavior with the **-startup** option, as described in “Specifying the Startup Tcl Script” on page 85.

You can use wildcard characters in your file names to specify multiple files. On Unix systems, the shell you use (that is, **sh**, **csh**, etc.) handles wildcard expansion. On Windows systems, **prowrap** uses Tcl’s **glob** command to handle wildcard expansion. (See the Tcl **glob** reference page for details of its operation.) So, in the above example, if *utils.tcl* and *help.tcl* were the only *.tcl* files in the *aux* subdirectory, you could accomplish the same effect as above with the following command:

```
C:> prowrap app.tcl aux\*.tcl
```

The files that you wrap are stored in the wrapped application’s internal file archive. For information on how pathnames are handled for wrapped files, see Table 6, “Resolving File Pathnames When Wrapping an Application” on page 87.

Specifying the Tcl Interpreter

By default, **prowrap** includes the **wish** Tcl interpreter, the [incr Tcl], [incr Tk], [incr Widget], TclX, and Expect (Unix only) extensions, and all of the binary libraries and library script files needed by **wish** and the extensions. The wrapped application is statically linked with all of the appropriate libraries, so it is not dependent on any other files; you can distribute it as a stand-alone application.

You can specify a different interpreter or different extension options with the **-uses** flag. For example, the following command includes the **tclsh** interpreter (with no extensions) and all of the binary libraries and library script files needed by **tclsh**:

```
C:> prowrap -uses tclsh app.tcl lib1.tcl lib2.tcl
```

The **-uses** flag is a convenience to simplify the use of certain standard configurations. Different **-uses** options provide predetermined sets of Tcl interpreters, extensions, and library files needed by the interpreter and extensions.

TclPro Wrapper then automatically includes all of those files with your wrapped application. Table 5 lists the values of **-uses** for which TclPro Wrapper has built-in support.

Table 5 Predefined **-uses** Options

Option	Description
bigwish (default)	Includes the **wish** Tcl interpreter, the [incr Tcl], [incr Tk], [incr Widget], TclX, and Expect (Unix only) extensions, and all of the library script files needed by **wish** and the extensions. Produces a statically-linked application.
bigtclsh	Includes the **tclsh** Tcl interpreter, the [incr Tcl], TclX, and Expect (Unix only) extensions, and all of the library script files needed by **tclsh** and the extensions. Produces a statically-linked application.
wish	Includes the **wish** interpreter (with no extensions) and all of the Tcl and Tk library script files. Produces a statically-linked application.
tclsh	Includes the **tclsh** interpreter (with no extensions) and all of the Tcl library script files. Produces a statically-linked application.
wish-dynamic	Includes the **wish** interpreter (with no built-in extensions), but not the Tcl or Tk library or library script files. Produces a dynamically-linked wrapped application, as discussed in “Creating and Distributing Dynamically-Linked Wrapped Applications” on page 92.
tclsh-dynamic	Includes the **tclsh** interpreter (with no built-in extensions), but not the Tcl library or library script files. Produces a dynamically-linked wrapped application, as discussed in “Creating and Distributing Dynamically-Linked Wrapped Applications” on page 92.

In addition to the options listed in Table 5, you can also define new configurations of your own with their own **-uses** values. See “Defining New -uses Options” on page 99 for details.

Specifying the Startup Tcl Script

By default, your wrapped application sources the first file you list in the **prowrap** command. You can use the **-startup** option to specify a different file to source when your application starts. This can be very helpful if you use wildcard characters to specify files to wrap. For example, consider the case of wrapping three Tcl scripts, *display.tcl*, *help.tcl*, and *main.tcl*, all in the same directory, and wanting to source *main.tcl* when you start your application. You could accomplish this with:

```
C:> prowrap -startup main.tcl *.tcl
```

You can create a wrapped application that displays an interactive Tcl shell by specifying the empty string (“”) as the **-startup** argument. Upon startup, the application doesn’t **source** any files automatically. Users can then access through the Tcl shell any additional files that you wrap with the application. For example:

```
C:> prowrap -uses tclsh -startup "" foo1.tcl foo2.tcl foo3.tcl
```

A user could then run the wrapped application and **source** *foo1.tcl*, *foo2.tcl*, or *foo3.tcl* from the Tcl shell as desired.

Passing Arguments to the Startup Tcl Script

With the **prowrap -arguments** option, you can specify additional arguments to your wrapped application that are treated just as if they were submitted to your unwrapped application on the command line. The arguments appear in the Tcl *argv* variable. The arguments you specify are inserted before any command-line arguments entered by the end user when they execute your wrapped application.

You must provide the arguments as a single argument on the **prowrap** command line; use proper quoting conventions of your command shell to accomplish this. For example, the following passes the arguments **-height 50 -width 20** to the *main.tcl* script:

```
c:> prowrap main.tcl img\*.gif -arguments "-height 50 -width 20"
```

Specifying the Name of a Wrapped Application

The default name of the wrapped application produced by **prowrap** is *prowrapout* on Unix or *prowrapout.exe* on Windows. You can use the **-out** option to specify a different name for the application. For example, the following creates a wrapped application with the name *myapp.exe*:

```
C:> prowrap myapp.tcl utils.tcl -out myapp.exe
```

Note On Windows systems, **prowrap** automatically adds the *.exe* extension if you omit it from the application name.

Determining Path References in Wrapped Applications

As discussed in “How the Internal File Archive Works in a Wrapped Application” on page 82, you must use relative pathnames to access files stored in the internal archive of a wrapped application. The proper pathname to use to access a file from the archive depends on your **prowrap** command arguments.

By default, files that you specify in your **prowrap** command with relative pathnames retain that pathname in the archive. Files that you specify with absolute pathnames are stripped of their drive and root directory characters. For example, consider in the following:

```
C:> prowrap myApp.tcl D:\tcl\common\utils.tcl
```

To source *D:\tcl\common\utils.tcl* from within your wrapped application, you would need to use a command such as:

```
source [file join tcl common utils.tcl]
```

You can also change the resulting pathname for a wrapped file with the **-relativeto** *directory* option to **prowrap**. The **-relativeto** flag instructs TclPro Wrapper to wrap all file name patterns that follow relative to the *directory* you specify. As an example, consider the following:

```
C:> prowrap myApp.tcl -relativeto D:\tcl\common \
    D:\tcl\common\utils.tcl
```

In this case, the resulting pathname for *D:\tcl\common\utils.tcl* from within your wrapped application is simply *utils.tcl*.

Table 6 summarizes how wrapped file pathnames are determined.

Table 6 Resolving File Pathnames When Wrapping an Application

Path Type	Using -relativeto flag?	Resulting Wrapped File Pathname	Example
Relative	No	The given relative pathname (including any "." or ".." relative pathname references)	*images/icon.gif* and *../lib/control.tcl* remain the same
Relative	Yes	The pathname of the file relative to the **-relativeto** directory	*images/icon.gif* with **-relativeto images** becomes *icon.gif* *../lib/control.tcl* with **-relativeto ../lib** becomes *control.tcl*
Absolute	No	The full pathname of the file without the root directory	*/usr/local/tcl/lib/common.tcl* becomes *usr/local/tcl/lib/common.tcl*
Absolute	Yes	The pathname of the file relative to the **-relativeto** directory	*/usr/local/tcl/lib/common.tcl* with **-relativeto /usr/local/tcl** becomes *lib/common.tcl*

Specifying TclPro Wrapper Command Line Arguments Using Standard Input

Many command shells have a limit to the number of characters they accept as input. Although this is rarely a problem when wrapping just a few Tcl scripts, you might exceed this limit if you use wildcard expansion and wrap lots of data files or Tcl packages.

To get around this limitation, **prowrap** allows you to specify arguments from standard input using the **-@** option. Arguments from standard input are processed after all other arguments on the **prowrap** command line.

Specifying Code to Execute at Application Startup

The **-code** option allows you to provide Tcl code that your application executes when it starts. The application executes the code early in the application initialization sequence, before **Tcl_Init** or any other package initialization procedures are invoked. You can specify multiple **-code** options, in which case TclPro Wrapper arranges for the application to execute these scripts in the order that they appear on the **prowrap** command line.

One common use for the **-code** option is to set the *auto_path* variable to handle Tcl script libraries wrapped with your applications. For example, the following **prowrap** command wraps an application with a library in the */usr/local/lib/common* directory and sets the *auto_path* variable so that the library is loaded properly on execution:

```
% prowrap myscript.tcl -relativeto /usr/local \
/usr/local/lib/common/*.tcl /usr/local/lib/common/tclIndex \
-code "lappend auto_path lib/common" -out myscript
```

Wrapping Libraries and Packages

Often, your application will use various Tcl libraries and packages. This section describes how to wrap libraries and packages with your application.

In this section, a *library* refers to either:

- A collection of Tcl scripts contained in a directory that also contains a *tclIndex* file generated by the **auto_mkindex** command
- A binary shared library that an application can load using the **load** command

In this book, a *package* refers to a collection of Tcl scripts or binary shared libraries in a directory that also contains a *pkgIndex.tcl* file generated by the **pkg_mkIndex** command.

Note You don't need to take any special steps to wrap applications that use the Tcl extensions bundled with TclPro (for example, [incr tcl]) if you specify the appropriate built-in **prowrap -uses** option. See "Specifying the Tcl Interpreter" on page 84 for more information.

Wrapping Libraries of Tcl Scripts

You must take special steps to auto-load Tcl script libraries that you wrap with your application. For example, if a library consists of the files *help.tcl* and *display.tcl*, and they and the *tclIndex* file are stored in */usr/local/lib/common*, an unwrapped Tcl script that used this library would contain the following command to auto-load the library:

```
lappend auto_path /usr/local/lib/common
```

This command would fail to auto-load your library in a wrapped application because of the absolute pathname. You can correct this problem in one of two ways:

- Change your application to test if it is executing as a wrapped application, and then set the *auto_path* variable appropriately:

  ```
  if {[info exists tcl_platform(isWrapped)]} {
      lappend auto_path lib/common
  } else {
      lappend auto_path /usr/local/lib/common
  }
  ```

  Then wrap your application as follows (remember to wrap the *tclIndex* file in addition to the Tcl script files):

  ```
  % prowrap myscript.tcl -relativeto /usr/local \
  /usr/local/lib/common/*.tcl /usr/local/lib/common/tclIndex
  ```

- Set the *auto_path* variable using the **-code** option of the **prowrap** command. The **-code** option executes the Tcl code that you provide before executing the Tcl scripts of your application. Thus, the following **prowrap** command accomplishes the same results as above (remember to wrap the *tclIndex* file in addition to the Tcl script files):

  ```
  % prowrap myscript.tcl -relativeto /usr/local \
  /usr/local/lib/common/*.tcl /usr/local/lib/common/tclIndex \
  -code "lappend auto_path lib/common"
  ```

Wrapping Binary Shared Libraries

Important Wrapped applications that **load** shared libraries must use a dynamically-linked Tcl interpreter such as **tclsh-dynamic** or **wish-dynamic**. If you use a statically-linked Tcl interpreter such as **tclsh** or **wish**, you will receive an error stating that the **load**

command is not supported when executing the wrapped application. For more information on selecting a Tcl interpreter for your wrapped application, see “Specifying the Tcl Interpreter” on page 84.

You can’t wrap binary shared libraries. There are two options for creating a wrapped application that uses a binary shared libraries:

- Create a custom Tcl interpreter that links a static version of the library.

Important TclPro Wrapper requires specially-written Tcl interpreters to work with wrapped applications. Any custom interpreters that you use must follow the guidelines described in “Creating Base Applications for TclPro Wrapper” on page 115.

- Wrap your application (without the binary shared library) with a dynamically-linked Tcl interpreter such as **tclsh-dynamic** or **wish-dynamic**. Then include the binary shared library in your distribution that you provide to customers. See “Creating and Distributing Dynamically-Linked Wrapped Applications” on page 92 for details.

Wrapping Tcl Script Packages

Packages that consist entirely of Tcl scripts don’t need any special handling when wrapping. TclPro Wrapper understands *pkgIndex.tcl* files and automatically adds wrapped directories to your application’s *tcl_pkgPath* variable if they contain *pkgIndex.tcl* files.

For example, if you have a package stored in */usr/local/lib/common* and you have generated a *pkgIndex.tcl* file in that directory using the **pkg_mkIndex** command, you can wrap the package automatically with a **prowrap** command such as:

```
% prowrap myscript.tcl -relativeto /usr/local \
/usr/local/lib/common/*.tcl -o myscript
```

Wrapping Packages Containing Binary Shared Libraries

You can’t wrap packages that contain binary shared libraries. There are two options for creating a wrapped application that uses packages with binary shared libraries:

- Create a custom Tcl interpreter that links a static version of the package.

Important TclPro Wrapper requires specially-written Tcl interpreters to work with wrapped applications. Any custom interpreters that you use must follow the guidelines described in “Creating Base Applications for TclPro Wrapper” on page 115.

- Wrap your application (without the packages) with a dynamically-linked Tcl interpreter such as **tclsh-dynamic** or **wish-dynamic**. Then include the packages in your distribution that you provide to customers. See “Creating and Distributing Dynamically-Linked Wrapped Applications” on page 92 for details.

Specifying a Temporary Directory

The **prowrap -temp** argument allows you to specify a directory that TclPro Wrapper uses to temporarily hold files created during the wrapping process. By default, TclPro Wrapper uses the directory given by either *TEMP*, *TMP*, or *TMPDIR* environment variables, which are checked in that order. On Unix, the directory falls through to the */tmp* directory if no environment variable exists.

For example, the following uses *C:\Temp* as a temporary directory for wrapping on a Windows system:

```
% prowrap -temp C:\Temp foo1.tcl foo2.tcl
```

Getting Detailed Wrapping Feedback

You can get TclPro Wrapper to give you more detailed information about what it is doing and which files it is wrapping by specifying the **prowrap -verbose** option.

Static and Dynamic Linking with Wrapped Applications

TclPro Wrapper allows you to create either statically-linked or dynamically-linked wrapped applications:

- A statically-linked application copies all the code it needs from libraries when you compile it. Once you compile the application, you no longer need the libraries to be able to run the application.
- A dynamically-linked application contains mechanisms for loading the code it needs from libraries as needed while the application is running. The application requires the libraries to be present while it runs so that it can dynamically load and execute the library code. On Windows, these libraries are usually referred to as *DLLs* (Dynamic Link Libraries). On Unix systems, they are often called *shared libraries*, because several application can use them at the same time.

Deciding Whether Static or Dynamic Linking is More Appropriate

In general, Scriptics recommends that you create statically-linked wrapped applications. A statically-linked application is usually simpler to distribute and maintain. It contains your scripts and data files, a Tcl interpreter, and everything else needed to run the application. On the other hand, if you distribute a dynamically-linked application, you must be sure that the target system has the Tcl (and Tk, if needed) libraries and library script files (such as *init.tcl*) properly installed and configured. If your application uses Tcl extensions (such as [incr Tcl]), then those extensions must also be installed and configured on your target system. Furthermore, if a user accidentally deletes a shared library, or another software package installs an incompatible version of one, your dynamically-linked application will no longer work on that system.

Important Because of system limitations, statically-linked wrapped applications can't load shared libraries. Therefore, if you need to **load** shared libraries (or auto-load packages that contain binary shared libraries), you must either create a dynamically-linked wrapped application or create a custom Tcl interpreter that links a static version of the library.

You also might consider distributing dynamically-linked wrapped applications. However, for a dynamically-linked wrapped application to work, your target systems must have all needed libraries installed and configured properly. Dynamically-linked applications are smaller than statically-linked ones, which can be beneficial if you plan to distribute several wrapped applications.

Creating and Distributing Dynamically-Linked Wrapped Applications

To create a dynamically-linked wrapped application, wrap your application with either the **-uses tclsh-dynamic** option (to use the **tclsh** interpreter) or **-uses wish-dynamic** option (to use the **wish** interpreter).

For a Windows application, if your target system has the same version of TclPro installed and your application doesn't use any extensions other than those bundled with TclPro, you can simply copy your application to the target system. You can run the application from anywhere on the target system.

For a Unix application, if your target system has the same version of TclPro installed *in exactly the same directory as on your development system* and your application doesn't use any extensions other than those bundled with TclPro, you can simply copy your application to the target system. You can run the application

from anywhere on the target system. Although the requirements in this case are restrictive, it is actually fairly common for a company to make the TclPro installation available on a shared directory of a file server. If all users mount the TclPro installation in the same location on their systems, they all effectively have the same TclPro configuration.

For all other cases, you must create a special distribution to install on your target system that contains your application and all binary libraries and library script files required by your system. (The rest of this section refers to this distribution directory as *$DIST*.) You must copy these files from the TclPro installation directory. (The rest of this section refers to this directory as *$TclPro*).

Your resulting distribution tree should have the following structure:

```
$DIST/
|
|---lib/
|   |
|   |---tcl8.2/
|   |---tk8.2/ (optional)
|   |---itcl3.1/ (optional)
|   |---itk3.1/ (optional)
|   |---iwidgets2.2.0/ (optional)
|   |---iwidgets3.0.0/ (optional)
|   |---tclX8.2/ (optional)
|   |---tkX8.2/ (optional)
|
|---$platform/
    |
    |---lib/ (Unix only)
    |   |
    |   |---*.so (or *.sl on HP-UX)
    |
    |---bin/
        |
        |---wrapped application(s)
        |---*.dll (Windows only)
```

The following steps describe how to create this distribution directory:

1) Create a dynamically-linked wrapped application with **prowrap**. The **-uses tclsh-dynamic** and **-uses wish-dynamic** options automatically handle setting the appropriate values of the *tcl_library* and *tk_library* variables, as well as any similar library variables for the extensions bundled with TclPro, so that your application can find the script libraries. If you use any additional extensions with your application, you must include a **-code** option to your **prowrap** command setting any similar library variables for those extensions. You can use the following example as a template:

```
-code "set tcl_library [file join [file dir [info nameofexec]] .. .. lib tcl8.2]"
```

You would need to replace “tcl_library” and “tcl8.2” with values appropriate for your extension.

2) Create a distribution directory with whatever name you want.

3) Create the directory *$DIST/lib*.

4) Copy the entire contents of *$TclPro/lib* to *$DIST/lib*. Optionally, you can omit from *$DIST/lib* any extensions your application doesn’t use (for example, don’t copy *$TclPro/lib/tclX8.2* and its contents if your application doesn’t use TclX).

5) If your application uses any additional extensions (beyond those bundled with TclPro) which have directories and files residing in the Tcl script library directory (the *lib* subdirectory of the Tcl installation directory), then copy those directories and files to *$DIST/lib*.

6) Create the directory *$DIST/$platform*, where *$platform* is the platform-specific subdirectory as used by TclPro. Table 7 lists the appropriate subdirectory names (for example, *win32-ix86* for Windows systems).

Table 7 Platform-Specific TclPro Subdirectories

Platform	TclPro Platform Subdirectory
HP-UX	*hpux-parisc*
IRIX/Mips	*irix-mips*
Linux/x86	*linux-ix86*
Solaris/SPARC	*solaris-sparc*
Windows 95/NT(x86)	*win32-ix86*

7) Create the directory *$DIST/$platform/bin*.

8) Copy or move your dynamically-linked wrapped application to *$DIST/$platform/bin.*

9) For Unix distributions:

   a) Create the directory *$DIST/$platform/lib.*

   b) Copy all shared libraries from *$TclPro/$platform/lib* to *$DIST/$platform/lib* (for example, copy *$TclPro/solaris-sparc/lib/*.so* to *$DIST/solaris-sparc/lib*). Optionally, you can omit from *$DIST/$platform/lib* any extensions your application doesn’t use (for example, don’t copy *$TclPro/$platform/lib/libtclx8.2.so* if your application doesn’t use TclX).

c) If your application uses any additional extensions (beyond those bundled with TclPro) which have shared libraries, or if your application uses any other shared libraries, then copy those libraries to *$DIST/$platform/lib*.

10) For Windows distributions:

a) Copy all shared libraries from *$TclPro\win32-ix86\bin* to *$DIST\win32-ix86\bin* (for example, copy *$TclPro\win32-ix86\bin*.dll* to *$DIST\win32-ix86\bin*). Optionally, you can omit from *$DIST\win32-ix86\bin* any extensions your application doesn't use (for example, don't copy *$TclPro\win32-ix86\bin\tclx805.dll* if your application doesn't use TclX).

b) If your application uses any additional extensions (beyond those bundled with TclPro) which have shared libraries, or if your application uses any other shared libraries, then copy those libraries to *$DIST\win32-ix86\bin*.

11) Use whatever installation method you want to copy the entire *$DIST* distribution tree to your target systems. You can install the distribution anywhere you like on the target system; however, users can't move the wrapped application from the distribution tree's *bin* directory.

Tip If you want to distribute more than one dynamically-linked wrapped application, you can include all of those applications in *$DIST/$platform/bin*. If you do this, be sure to include *all* of the extensions and libraries needed by *all* of your applications.

Wrapping Applications with a Custom Interpreter or Custom Initialization Libraries

You can wrap an application with a Tcl interpreter other than those supported by the built-in **prowrap -uses** options. You can also wrap an application that uses a custom Tcl initialization library (that is, *init.tcl*). You can specify custom interpreters on either an as-needed basis or, if you frequently use the same interpreter, you can create your own custom **-uses** option.

Important Only specially-written Tcl interpreters work with wrapped applications. The built-in **prowrap -uses** options automatically use supported Tcl interpreters. However, any custom interpreters that you use must follow the guidelines described in "Creating Base Applications for TclPro Wrapper" on page 115.

Specifying a Custom Interpreter or Custom Initialization Files

The **prowrap -executable** option specifies a Tcl interpreter to wrap with your application. If you include both the **-uses** and **-executable** options when wrapping, TclPro Wrapper automatically wraps all the initialization library files required for the standard interpreter (for example, *init.tcl*), but includes the custom interpreter you specify rather than the standard interpreter.

The **prowrap -tcllibrary** option specifies where the wrapped application can find the Tcl initialization library files *at the time that it is run* (rather than at the time you wrap the application). In other words, it sets the value of the Tcl *tcl_library* variable used by your application during its initialization. You don't need to include the **prowrap -tcllibrary** option if your application uses the standard initialization files and you wrap the application using a built-in **-uses** option. The built-in **-uses** options automatically wrap the standard initialization files and sets the *tcl_library* variable appropriately.

Important As with any other file reference in a wrapped application, the **prowrap -tcllibrary** value you specify can refer to either a file on the disk or a file in the internal file archive. In other words, if you provide an absolute pathname, your application looks for the initialization files on the disk when it runs. On the other hand, if you specify a relative pathname, your application first looks in its internal file archive for the initialization files, and looks on the disk only if the files don't exist in the archive. For more information on file access in a wrapped application, see "How the Internal File Archive Works in a Wrapped Application" on page 82.

Important Always use the **prowrap -tcllibrary** option to set the value of the Tcl *tcl_library* variable rather than setting it using a **prowrap -code** option. Your application needs the correct value of the Tcl *tcl_library* variable during the initialization of the Tcl interpreter (primarily to find the character encoding files). Code that you include with the **-code** option is executed after initialization of the core Tcl interpreter.

Note The **prowrap -tcllibrary** option does not set the value of the Tcl *tk_library* variable or any other similar variable used by a Tcl package. Use the **prowrap -code** option to set these variables if you don't use a built-in **-uses** option.

The proper combination of **-uses**, **-executable**, and **-tcllibrary** options depends on:

- Whether you are creating a statically- or dynamically-linked application
- Whether or not you use a custom interpreter
- Whether or not you use custom initialization files

Obviously, if you use a standard interpreter and standard initialization file, you can simply use the built-in **-uses** options as described in "Specifying the Tcl Interpreter" on page 84. The other cases are described in the following sections.

Creating a Statically-Linked Wrapped Application with a Custom Interpreter and Standard Initialization Files

When creating a statically-linked wrapped application with a custom interpreter and the standard Tcl initialization files, your **prowrap** command line must include the following:

- One of the standard **-uses** options
- An **-executable** option specifying the custom interpreter

For example, the following wraps an application with a custom interpreter, **myWish**, that is based on the standard **wish** interpreter:

```
% prowrap -uses wish -executable /usr/local/bin/myWish \
-out myApp myApp.tcl img/*.gif
```

Creating a Statically-Linked Wrapped Application with a Standard Interpreter and Custom Initialization Files

When creating a statically-linked wrapped application with a standard interpreter and custom Tcl initialization files, your **prowrap** command line must include the following:

- One of the standard **-uses** options
- All of your custom Tcl initialization files
- A **-tcllibrary** option specifying the location of the initialization files in the wrapped application's file archive
- Any **-code** options required to initialize other Tcl configuration variables such as *tk_library*

For example, the following wraps an application with the standard **wish** interpreter and a set of initialization files contained in the directory */usr/local/siteTcl/lib*:

```
% prowrap -uses wish -out myApp myApp.tcl img/*.gif \
/usr/local/siteTcl/lib/tcl8.2/*.tcl \
/usr/local/siteTcl/lib/tcl8.2/tclIndex \
/usr/local/siteTcl/lib/tcl8.2/encoding/*.enc \
/usr/local/siteTcl/lib/tk8.2/*.tcl \
/usr/local/siteTcl/lib/tk8.2/tclIndex \
-tcllibrary usr/local/siteTcl/lib/tcl8.2 \
-code "set tk_library usr/local/siteTcl/lib/tk8.2"
```

Note Both the **-tcllibrary** and **-code** options omit the initial "/" when specifying the pathnames for the *tcl_library* and *tk_library* variables. This is because TclPro Wrapper strips the initial "/" from absolute pathnames when wrapping files, and therefore the wrapped initialization files have relative pathnames in the archive (for example, "*usr/local/siteTcl/lib/tcl8.2/init.tcl*"). See "How the Internal File Archive Works in a Wrapped Application" on page 82 for more information.

Creating a Statically-Linked Wrapped Application with a Custom Interpreter and Custom Initialization Files

When creating a statically-linked wrapped application with a custom interpreter and custom Tcl initialization files, you are basically overriding all TclPro Wrapper defaults and wrapping your application "from scratch." In this case, your **prowrap** command line must include the following:

- The **-uses ""** option to prevent TclPro Wrapper from using any built-in **-uses** configuration
- A **-executable** option specifying the custom interpreter
- All of your custom Tcl initialization files
- A **-tcllibrary** option specifying the location of the initialization files in the wrapped application's file archive
- Any **-code** options required to initialize other Tcl configuration variables such as *tk_library*

For example, the following wraps an application with a custom interpreter, **myWish**, that is based on the standard **wish** interpreter and a set of initialization files contained in the directory */usr/local/siteTcl/lib*:

```
% prowrap -uses "" -executable /usr/local/bin/myWish \
-out myApp myApp.tcl img/*.gif \
/usr/local/siteTcl/lib/tcl8.2/*.tcl \
/usr/local/siteTcl/lib/tcl8.2/tclIndex \
/usr/local/siteTcl/lib/tcl8.2/encoding/*.enc \
/usr/local/siteTcl/lib/tk8.2/*.tcl \
/usr/local/siteTcl/lib/tk8.2/tclIndex \
-tcllibrary usr/local/siteTcl/lib/tcl8.2 \
-code "set tk_library usr/local/siteTcl/lib/tk8.2"
```

Note Both the **-tcllibrary** and **-code** options omit the initial "/" when specifying the pathnames for the *tcl_library* and *tk_library* variables. This is because TclPro Wrapper strips the initial "/" from absolute pathnames when wrapping files, and therefore the wrapped initialization files have relative pathnames in the archive (for example, "*usr/local/siteTcl/lib/tcl8.2/init.tcl*"). See "How the Internal File Archive Works in a Wrapped Application" on page 82 for more information.

Creating a Dynamically-Linked Wrapped Application with a Custom Interpreter

When creating a dynamically-linked wrapped application, your application depends on all shared libraries and library script files (for example, *init.tcl*) already being installed and configured on your target system. The built-in **-uses tclsh-dynamic** and **-uses wish-dynamic** options automatically handle setting the appropriate values of the *tcl_library* and *tk_library* variables, as well as any similar library variables for the extensions bundled with TclPro. But when you want to use a custom dynamically-linked interpreter, you must set these values yourself when wrapping the application. In this case, your **prowrap** command line must include the following:

- The **-uses ""** option to prevent TclPro Wrapper from using any built-in **-uses** configuration
- A **-executable** option specifying the custom interpreter
- A **-tcllibrary** option specifying the location of the initialization files on your target system
- Any **-code** options required to initialize other Tcl configuration variables such as *tk_library*

Furthermore, you must configure your target systems as discussed in “Creating and Distributing Dynamically-Linked Wrapped Applications” on page 92 (including creating a distribution directory as described in that section, if necessary).

For example, the following wraps an application with a custom dynamically-linked interpreter, **myWish**, that is based on the standard **wish** interpreter. The example assumes that you create a distribution directory for your application as described in “Creating and Distributing Dynamically-Linked Wrapped Applications”:

```
% prowrap -uses "" -executable /usr/local/bin/myWish \
-out myApp myApp.tcl img/*.gif -tcllibrary ../../lib/tcl8.2 \
-code "set tk_library [file join [file dir [info nameofexec]] .. .. lib tk8.2] "
```

Defining New -uses Options

TclPro Wrapper recognizes files with the *.uses* extension as providing additional **-uses** configurations. For example, a file *new.uses* directory defines a configuration named “new” that you can use as a **-uses** option.

When you specify a **-uses** option, TclPro Wrapper checks to see if it is a built-in configuration first. If not, if looks for a *.uses* file with the proper name in the *lib/prowrapuses* directory of the TclPro installation (that is, *lib/prowrapuses*

should be at the same level as the *lib/tcl8.2* directory). If TclPro Wrapper doesn't find the proper file there, it finally checks the directory from which you execute **prowrap**.

You can also specify an absolute or relative path as an argument to the **-uses** option. For example, specifying **-uses C:\Tcl\Wrapper\custom** causes TclPro Wrapper to use the configuration file *C:\Tcl\Wrapper\custom.uses*.

The *lib/prowrapuses* directory of the TclPro installation contains Tcl scripts showing the definitions of the built-in **-uses** options. You can use these files as templates for creating your own **-uses** configurations.

Note Modifying these files does not change the behavior of the built-in **-uses** configurations; they are only sample files. To use them, you can copy them, rename them, and modify them as needed.

TclPro Wrapper evaluates the contents of a *.uses* file when it prepares to wrap an application with that configuration. The *.uses* file must contain a Tcl script that returns a Tcl list providing additional TclPro Wrapper command-line arguments. These arguments should typically specify the following:

- an **-executable** option specifying a Tcl or interpreter
- if this option produces statically-wrapped applications, all initialization and support files required by the interpreter (for example, the contents of the Tcl and Tk *lib* directories and their subdirectories)
- a **-tcllibrary** option specifying the location Tcl initialization library files (that is, the value of the Tcl *tcl_library* variable)
- if the option includes a Tk interpreter, a **-code** option setting the value of the *tk_library* variable
- if this option provides built-in support for additional Tcl libraries or packages, the script and index files for these packages as discussed in "Wrapping Libraries and Packages" on page 88
- optionally, one or more **-code** options to perform any other required initialization of a wrapped application (for example, setting any required values for an included package)
- optionally, any other desired script or data files

Important As with any other file reference in a wrapped application, the file references you provide to the **-tcllibrary** and **-code** options can refer to either a file on the disk or a file in the internal file archive. In other words, if you provide an absolute pathname, your application looks for the initialization files on the disk when it runs. On the other hand, if you specify a relative pathname, your application first looks in its internal file archive for the initialization files, and looks on the disk only

if the files don't exist in the archive. Also remember that TclPro Wrapper strips the initial "/" from absolute pathnames when wrapping files, and therefore wrapped initialization files have relative pathnames in the archive. For more information on file access in a wrapped application, see "How the Internal File Archive Works in a Wrapped Application" on page 82.

For example, suppose you create a custom, statically-linked Tcl interpreter with the name *siteTclsh1.0* and place it in the directory */usr/local/tcl/site1.0/bin*. In addition to the standard Tcl script library files, located in */usr/local/tcl/lib/tcl8.2*, your custom interpreter requires the custom initialization and support files *site.tcl*, *siteApp.tcl*, and *help.txt*, which you place in the directory */usr/local/tcl/site1.0/lib*. Your custom interpreter uses a custom Tcl variable, *site_library*, to locate its initialization and support files. To define this interpreter and support files as a custom **-uses** option named "siteTclsh", create the file *siteTclsh.uses* and place it in the *lib/prowrapuses* directory. The *siteTclsh.uses* file would contain:

```
# siteTclsh.uses
return [list \
    -executable /usr/local/tcl/site1.0/bin/siteTclsh1.0 \
    -relativeto /usr/local/tcl \
    /usr/local/tcl/lib/tcl8.2/*.tcl \
    /usr/local/tcl/lib/tcl8.2/tclIndex \
    /usr/local/tcl/lib/tcl8.2/encoding/*.enc \
    /usr/local/tcl/site1.0/lib/site.tcl \
    /usr/local/tcl/site1.0/lib/siteApp.tcl \
    /usr/local/tcl/site1.0/lib/help.txt \
    -tcllibrary lib/tcl8.2 \
    -code "set site_library [file join site1.0 lib]" ]
```

You could then wrap applications using this custom shell by specifying the **-uses siteTclsh** option. For example, the following TclPro Wrapper command would create a wrapped application based on *siteTclsh1.0* with *file1.tcl* as the startup script:

```
% prowrap -uses siteTclsh file1.tcl file2.tcl
```

Preparing an Application for Wrapping

There are minor differences in the way an application runs when it is wrapped versus when it runs unwrapped. However, it is relatively easy to modify your application so that you can test it in unwrapped form, then wrap the same files for distribution. This section shows you how to change your application to ensure that it works properly both unwrapped and wrapped.

Detecting When an Application Is Wrapped

Because there are minor differences in the behavior of unwrapped and wrapped applications, you need to be able to detect whether your application is wrapped or not. TclPro Wrapper automatically creates the variable *tcl_platform(isWrapped)* when it wraps your application, so your application simply needs to test for the existence of this variable to determine whether or not it is wrapped. The following code fragment demonstrates how to use *tcl_platform(isWrapped)*:

```
if {[info exists tcl_platform(isWrapped)]} {
    # Application is wrapped
} else {
    # Application is not wrapped
}
```

Modifying Custom Shells

TclPro Wrapper requires specially-written Tcl interpreters to work with wrapped applications. The predefined **prowrap -uses** options (described in “Specifying the Tcl Interpreter” on page 84) automatically use appropriate interpreters. However, if you want your application to use a custom interpreter, you use must write that interpreter following the guidelines in “Creating Base Applications for TclPro Wrapper” on page 115.

Changing File References

Writing an application to work properly both unwrapped and wrapped can be tricky when it comes to file access. You want to prevent accidental fall-through and file shadowing, as discussed in “How the Internal File Archive Works in a Wrapped Application” on page 82. The key points to keep in mind are:

- All files in the internal archive of a wrapped application have relative pathnames
- If you use the **-relativeto** option when wrapping a file, the pathname of a file in the internal archive is different from its corresponding unwrapped pathname (see “Determining Path References in Wrapped Applications” on page 86)
- A wrapped application always searches for a file in its internal file archive before searching the disk whenever it encounters a relative pathname to a file

Accessing Unwrapped Files

If your wrapped application attempts to access unwrapped files using relative pathnames, it runs the risk of accidentally accessing a file in the internal archive instead (that is, file shadowing). To ensure that your application always accesses unwrapped files when desired, you should always use absolute pathnames in a wrapped application.

In particular, you should be careful in how your application handles cases where a user can enter a file name. If a user enters a relative pathname for a file, you should convert it to an absolute pathname. For example, if the variable *path* contains a relative file name, you can create an absolute file name by appending it to the current working directory:

```
set path [file join [pwd] $path]
```

Accessing Files from a Shared Directory

Files shared by multiple applications or projects are typically put in a shared directory, often on a file server. An application *myscript.tcl* might then access those files as follows:

```
set shared {Z:\tcl\common}
source [file join $shared help.tcl]
source [file join $shared display.tcl]
```

Unfortunately, because of the absolute pathname, the code above no longer works if you wrap the files in the shared directory with the application.

However, you can easily modify this code to work either unwrapped or wrapped by testing to see whether the application is wrapped and modifying the value of *shared* appropriately. For example:

```
if {[info exist tcl_platform(isWrapped)]} {
    set shared common
} else {
    set shared {Z:\tcl\common}
}
source [file join $shared help.tcl]
source [file join $shared display.tcl]
```

You would then need to wrap the shared files using the **-relativeto** flag as in the following example:

```
C:> prowrap myscript.tcl -relativeto Z:\tcl Z:\tcl\common\*.tcl
```

Accessing Wrapped Files Relative to a Script's Directory

A common trick to avoid hard-wiring pathnames into scripts is to figure out where the script is located with the **info script** command and then accessing files relative to the script's directory. For example:

```
set home [file dirname [info script]]
source [file join $home help.tcl]
source [file join $home display.tcl]
```

Auto-Loading Wrapped Tcl Script Libraries

You must take special steps to auto-load Tcl script libraries that you wrap with your application. “Wrapping Libraries of Tcl Scripts” on page 89 describes the changes you need to make to your application.

Changing the Windows Icon for a Wrapped Application

On Windows, a wrapped application receives the same icon as that of the Tcl interpreter that you wrap with the application. You can use a commercial or shareware icon manager to change the icon. You can also use Microsoft Visual C++ 5.0 or later on a Windows NT system to change the icon. (However, Microsoft Visual C++ on a Windows 95/98 system does not provide this feature.)

Important You should always change the icon of the Tcl interpreter before wrapping rather than attempting to change the icon of the final wrapped application executable. The wrapped application executable contains a Zip-formatted archive of the wrapped script and data files appended to the base interpreter. When a program attempts to change the icon of a wrapped application, it can become confused by the appended Zip archive and overwrite or destroy information contained in the archive.

The standard set of Tcl interpreters used by TclPro Wrapper to create wrapped applications are stored in the *win32-ix86\lib* subdirectory of your TclPro installation (for example, *C:\Program Files\TclPro1.3\win32-ix86\lib*). The name of each file and its corresponding **prowrap -uses** option is shown in Table 8.

Table 8 Tcl Interpreters Corresponding to **prowrap -uses** Options

Interpreter File Name	Corresponding prowrap -uses Option
wrapbigwish82s.in	**-uses bigwish** (default)
wrapbigtclsh82s.in	**-uses bigtclsh**
wrapwish82s.in	**-uses wish**
wraptclsh82s.in	**-uses tclsh**
wrapwish82.in	**-uses wish-dynamic**
wraptclsh82.in	**-uses tclsh-dynamic**

Important Always create a backup of any interpreter before attempting to change its icon. If you accidentally damage an interpreter and don't have a backup, you will have to reinstall TclPro to restore the interpreter.

If you change the icon for one of these interpreters, then all wrapped applications that you create using that interpreter inherit that icon. Another option is to create a copy of an interpreter, change the icon for that interpreter, and then use the **prowrap -executable** option so that TclPro Wrapper uses that interpreter when creating your wrapped application.

For example, if you have created a copy of the **wish** interpreter named *mywish.in* with a customized icon, you could use it when wrapping an application as follows:

```
C:> prowrap -uses wish -executable C:\Program
Files\TclPro1.3\win32-ix86\lib\mywish.in myApp.tcl
```

As mentioned earlier, you can use a commercial or shareware icon manager to change the icon. You can also use Microsoft Visual C++ 5.0 or later on a Windows NT system by performing the following steps:

1) Close any open workspaces.
2) Open the wrapped application in Visual C++.
3) Select File | Open from the menubar and browse for the wrapped application executable.
4) Select File of Type: Executable.
5) Select Open as: Resources. A window appears with the executable's resources and the name of the application in the title bar.
6) Click the icon folder to display the application's default icon.
7) Open the icon resource.
8) A window appears with the application icon.
9) Draw or paste your application icon.
10) Save the executable with your changes. Select File | Save from the menubar.
11) Create a shortcut on your Windows desktop to see your change.

Chapter 7
Creating Custom Interpreters with TclPro

This chapter describes how to create both regular Tcl interpreters and Tcl interpreters that you can use with the TclPro Wrapper. In general, you create Tcl interpreters with TclPro just as you would with the free Tcl distribution. However, TclPro makes it easier to build custom Tcl interpreters by providing precompiled libraries for Tcl, Tk, and all bundled extensions on each platform supported by TclPro. TclPro also provides libraries that support the **tbcload** extension, which is required to read the bytecode files created by TclPro Compiler, and the TclPro Wrapper library, which you need to create interpreters (that is, *base applications*) for use by TclPro Wrapper.

Important The development libraries and other files described in this chapter are part of the TclPro "C Development Libraries" installation component. You must install the TclPro "C Development Libraries" component if you want to use these files to create custom Tcl interpreters.

Remember, there is often no need for you to create a custom Tcl interpreter. If all you want to do is to incorporate a new extension, it is usually easier to use the built-in **load** and **package** facilities of Tcl. Also remember that **protclsh** or **prowish** already have built-in support for the extensions bundled with TclPro.

Note This chapter assumes that you are already familiar with writing custom Tcl interpreters; therefore, it concentrates on describing the unique features of building a custom Tcl interpreter with the TclPro distribution. For detailed instructions on writing a custom Tcl interpreter, consult the references listed in "For More Information" on page 4.

Overview of the TclPro Development Environment

This section provides general information about the TclPro development environment including the location of the libraries and sample files, and special comments about the compilation options of the Windows libraries.

Locations of the Libraries

All of the precompiled libraries shipped with TclPro are located in subdirectories of the TclPro installation directory. The libraries are organized by platform, with directory names as shown in Table 9.

Table 9 Locations of TclPro Libraries Relative to the Installation Directory

Platform	Library Subdirectory
HP-UX	*hpux-parisc/lib*
IRIX/Mips	*irix-mips/lib*
Linux/x86	*linux-ix86/lib*
Solaris/SPARC	*solaris-sparc/lib*
Windows 95/NT(x86)	*win32-ix86\lib* (static and export libraries) *win32-ix86\bin* (dynamic libraries)

For example, if you install TclPro in *C:\Program Files\TclPro1.3*, the static and export Windows libraries are in *C:\Program Files\TclPro1.3\win32-ix86\lib*.

Debug and Non-Debug Libraries for Windows

TclPro includes both debug and non-debug versions of all Windows libraries shipped. If you compile your application with debug options, you should be certain to link with libraries compiled with compatible debug options so that you can properly debug your extensions.

The Windows libraries shipped with TclPro are compiled with Visual C++ with the following compilation flags:

/MD Dynamic library, no debug

/MDd Dynamic library with debug

/MT Static library, no debug

/MTd Static library with debug

You should compile and link all components of your application with consistent compilation settings. To set these compilation flags in a Visual C++ Developer Studio project, display the Project Settings dialog, select the C/C++ tab, and select the Code Generation category. The compilation flags mentioned above correspond to the following Use Run-time Library selections:

/MD	Multithreaded DLL
/MDd	Debug Multithreaded DLL
/MT	Multithreaded
/MTd	Debug Multithreaded

The Sample Application

The TclPro "C Development Libraries" component installs a sample application in the *demos/sampleApp* subdirectory of the TclPro installation directory (for example, *C:\Program Files\TclPro1.3\demos\sampleApp*). Refer to the README file in that directory for a description of its contents.

The directory includes a Makefile for creating statically- and dynamically-linked versions of a sample interpreter and wrapped application. The applications demonstrate many of the topics discussed in this chapter. You might find the Makefile and the source files it compiles useful templates for creating your own custom interpreter.

Creating Regular Tcl Interpreters

You write Tcl interpreters with the TclPro development environment just as you would with the free Tcl distribution. Simply implement **Tcl_AppInit** to perform whatever initialization your application requires, and call **Tcl_Main** or **Tk_Main** from your main program. Then compile your program and link with either the static or dynamic libraries as appropriate.

Important TclPro Wrapper requires specially-written Tcl interpreters to work with wrapped applications. Any custom interpreters that you use with TclPro Wrapper must follow the guidelines described in "Creating Base Applications for TclPro Wrapper" on page 115.

Creating Statically-Linked Interpreters

When writing a statically-linked interpreter, you must explicitly initialize all extensions that are statically linked with your application. Typically, this consists of adding code to your application's **Tcl_AppInit** procedure calling the extension's **Init** procedure and then calling **TclStaticPackage** to register the extension as a statically-linked package. Then at compilation, you must link your application with static versions of every library that your application needs.

For example, the main source file for a Tcl application that is statically linked with **tbcload** and [incr Tcl] contains code similar to the following that shown below. You would then need to link this application with the **tbcload** and [incr Tcl] libraries in addition to the Tcl library.

```
#include "tcl.h"
.
.
.
static int MyAppInit(Tcl_Interp *interp);
int
main(argc, argv)
    int argc; /* Number of command-line arguments. */
    char **argv; /* Values of command-line arguments. */
{
    Tcl_Main(argc, argv, MyAppInit);

    return 0; /* Needed only to prevent compiler warning. */
}
static int
MyAppInit(interp)
    Tcl_Interp *interp; /* Interpreter for application. */
{
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Tbcload_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    Tcl_StaticPackage(interp, "tbcload", Tbcload_Init,
        Tbcload_SafeInit);
    if (Itcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    Tcl_StaticPackage(interp, "Itcl", Itcl_Init, Itcl_SafeInit);
.
.
.
    return TCL_OK;
}
```

Statically Linking Windows Interpreters

To create a statically-linked Tcl application under Windows, you link your application with the static version of the Tcl library and, if needed, the Tk library. You also must link with all other Tcl extension libraries used by your application, and any other application-specific libraries your application might use (for example, custom driver software for interacting with a special peripheral device).

The TclPro static Windows libraries are located in the *win32-ix86\lib* subdirectory of the TclPro installation directory. Table 10 lists the static libraries shipped with TclPro.

Table 10 Windows Libraries for Static Linking

Library	Description
tcl82s.lib	Tcl static library without debugging symbols
tcl82sd.lib	Tcl static library with debugging symbols
tk82s.lib	Tk static library without debugging symbols
tk82sd.lib	Tk static library with debugging symbols
tbcload13s.lib	tbcload static library without debugging symbols
tbcload13sd.lib	tbcload static library with debugging symbols
itcl31s.lib	[incr Tcl] static library without debugging symbols
itcl31sd.lib	[incr Tcl] static library with debugging symbols
itk31s.lib	[incr Tk] static library without debugging symbols
itk31sd.lib	[incr Tk] static library with debugging symbols
tclx82s.lib	TclX without debugging symbols
tclx82sd.lib	TclX with debugging symbols
tkx82s.lib	TkX without debugging symbols
tkx82sd.lib	TkX with debugging symbols

Note that TclPro uses the convention of ending a static library with the letter “s”; this makes it easy to distinguish *.lib* files that are export libraries for a dynamic library from corresponding static libraries. For example, *tcl82.lib* is the export library for *tcl82.dll*, whereas *tcl82s.lib* is the Tcl static library. Note also that the “d” convention is used as well, so that *tcl82sd.lib* is a static library built with debug options. The “d” libraries were all built with the **/MTd** flag, the others with **/MT**.

If you use the “d” libraries, link your application with *LIBCMTD.LIB*; otherwise link it with *LIBCMT.LIB*.

Statically Linking Unix Interpreters

To create a statically-linked Tcl application under Unix, you link your application with the static version of the Tcl library and, if needed, the Tk library. You also must link with all other Tcl extension libraries used by your application, and any other application-specific libraries your application might use (for example, custom driver software for interacting with a special peripheral device). Unlike Windows, there are no separate debug and non-debug libraries.

All of the Unix libraries shipped with TclPro are located in subdirectories of the TclPro installation directory. The libraries are organized by platform, with directory names as shown in Table 9 on page 108. Table 11 lists the static libraries shipped for Unix systems.

Table 11 Unix Libraries for Static Linking

Unix Library	Description
libtcl8.2.a	Tcl static library
libtk8.2.a	Tk static library
libtcload13s.a	tbcload static library
libitcl31s.a	[incr Tcl] static library
libitk31s.a	[incr Tk] static library
libtclx8.2.a	TclX static library
libtkx8.2.a	TkX static library
libexpect5.31.a	Expect static library

Note The IRIX libraries are compiled with the **-n32** flag.

Note that many of the static libraries end with the letter “s”; this is especially useful in that it eliminates some ambiguities in the interpretation of **-l** linker flags. For example, **-ltbcload13** refers to the shared library implementation of **tbcload**, whereas **-ltbcload13s** refers to the static version. If the “s” convention were not used, the **-l** flag for either would be **-ltbcload13**, and which one of the two libraries is used for the linking would depend on the resolution rules currently active in the linker.

Creating Dynamically-Linked Interpreters

Aside from implementing either **Tcl_AppInit** or **Tk_AppInit** as appropriate and calling **Tcl_Main** or **Tk_Main** from your main program, there are no special requirements for writing a dynamically-linked interpreter. Because your application automatically loads extension libraries as needed, you don't need to register them with **Tcl_StaticPackage** or initialize them with their corresponding **Init** procedures.

Dynamically Linking Windows Interpreters

To create a dynamically-linked Tcl application under Windows, you link your application with *export* libraries, which have a *.lib* extension. At run-time, your application loads the dynamic library corresponding to that export library, which has the same name as the export library but a *.dll* extension. You don't need the *.lib* file at run-time.

You link your application with the appropriate Tcl library and, if needed, the appropriate Tk library. You don't need to link with any other Tcl extension libraries; your application loads the dynamic libraries for any other extensions as needed at run-time. You must also link with any other application-specific libraries your application might use (for example, custom driver software for interacting with a special peripheral device).

The TclPro Windows libraries are located in subdirectories of *win32-ix86* in the TclPro installation directory. Table 12 lists the dynamic libraries shipped and their corresponding locations.

Table 12 Windows Libraries for Dynamic Linking

Dynamic Library	Export Library	Description
bin\tcl82.dll	*lib\tcl82.lib*	Tcl without debugging symbols
bin\tcl82d.dll	*lib\tcl82d.lib*	Tcl with debugging symbols
bin\tk82.dll	*lib\tk82.lib*	Tk without debugging symbols
bin\tk82d.dll	*lib\tk82d.lib*	Tk with debugging symbols
bin\tbcload13.dll	*lib\tbcload13.lib*	tbcload without debugging symbols
bin\tbcload13d.dll	*lib\tbcload13d.lib*	tbcload with debugging symbols
bin\itcl31.dll	*lib\itcl31.lib*	[incr Tcl] without debugging symbols
bin\itcl31d.dll	*lib\itcl31d.lib*	[incr Tcl] with debugging symbols

Table 12 Windows Libraries for Dynamic Linking *(Continued)*

Dynamic Library	Export Library	Description
bin\itk31.dll	*lib\itk31.lib*	[incr Tk] without debugging symbols
bin\itk31d.dll	*lib\itk31d.lib*	[incr Tk] with debugging symbols
bin\tclx82.dll	*lib\tclx82.lib*	TclX without debugging symbols
bin\tclx82d.dll	*lib\tclx82d.lib*	TclX with debugging symbols
bin\tkx82.dll	*lib\tkx82.lib*	TkX without debugging symbols
bin\tkx82d.dll	*lib\tkx82d.lib*	TkX with debugging symbols

Note that TclPro uses the convention of ending the name of a library that was built with debugging options with the letter “d.” For example, *tcl82d.dll* is the Tcl DLL built with debugging turned on and *tcl82d.lib* is its export library. The “d” libraries were all built with the **/MDd** flag, the others with **/MD**.

If you use the debug libraries, also link your application with *MSVCRTD.LIB*; otherwise link it with *MSVCRT.LIB*.

Dynamically Linking Unix Interpreters

To create a dynamically-linked Tcl application under Unix, you link your application directly with the appropriate shared libraries. Unlike Windows, there are no export libraries, and you don’t need separate debug and non-debug libraries.

You link your application with the appropriate Tcl library and, if needed, the appropriate Tk library. You don’t need to link with any other Tcl extension libraries; your application loads the dynamic libraries for any other extensions as needed at run-time. You must also link with any other application-specific libraries your application might use.

All of the Unix libraries shipped with TclPro are located in subdirectories of the TclPro installation directory. The libraries are organized by platform, with directory names as shown in Table 9 on page 108. Table 13 lists the shared libraries shipped for Unix systems.

Table 13 Unix Libraries for Dynamic Linking

Unix Library	Description
libtcl8.2.so (*libtcl8.2.sl* on HP-UX)	Tcl shared library
libtk8.2.so (*libtk8.2.sl* on HP-UX)	Tk shared library
libtcload13.so (*libtcload13.sl* on HP-UX)	tbcload shared library
libitcl31.so (*libitcl31.sl* on HP-UX)	[incr Tcl] shared library
libitk31.so (*libitk31.sl* on HP-UX)	[incr Tk] shared library
libtclx8.2.so (*libtclx8.2.sl* on HP-UX)	TclX shared library
libtkx8.2.so (*libtkx8.2.sl* on HP-UX)	TkX shared library
libexpect5.31.so (*libexpect5.31.sl* on HP-UX)	Expect shared library

Note The IRIX libraries are compiled with the **-n32** flag.

Creating Base Applications for TclPro Wrapper

This section describes how to create a Tcl interpreter that you can use with TclPro Wrapper, otherwise known as a *base application*. Base applications require special support for accessing files from the wrapped application's internal file archive.

Note You can also use a base application as a regular Tcl interpreter for an unwrapped applications.

In general, writing a base application is the same as writing a regular Tcl interpreter. Typically, the only changes you have to make are:

- Include *proWrap.h* in your application (*proWrap.h* is located in the *include* subdirectory of the TclPro installation directory)
- Call **Pro_WrapTclMain** or **Pro_WrapTkMain** from your application instead of **Tcl_Main** or **Tk_Main**
- Compile *proWrapTclMain.c* or *proWrapTkMain.c* (which are located respectively in the *lib/tcl8.2* and *lib/tk8.2* subdirectories of the TclPro installation directory) and link your application with the resulting object file
- Link your application with the appropriate TclPro Wrapper library

Other than these changes, you write your base application as you would a regular interpreter and link it with all other libraries you would typically need to link with (for example, *tcl82s.lib*, *tbcload13s.lib*, *itcl31s.lib*, etc.). See the appropriate section of "Creating Regular Tcl Interpreters" for detailed instructions.

Unless you have modified the standard **Tcl_Main** or **Tk_Main** procedures, you should not have to make any changes to the *proWrapTclMain.c* or *proWrapTkMain.c* files. See "Modifying the Base Application Default Main Files" on page 118 if you need to make changes to these files or write your own versions of these procedures.

Note File access functions in the Tcl and Tk C libraries (for example, **Tcl_OpenFileChannel** and **Tk_GetBitmap**) access files in the internal archive of a wrapped application in the same manner as file access procedures in Tcl scripts (for example, **source** and **open**). See "How the Internal File Archive Works in a Wrapped Application" on page 82 for more information on the internal file archive of a wrapped application.

Tip If you are writing a new interpreter, you can use the files listed in Table 14 as templates for your interpreter's **main** and **Tcl_AppInit** procedures. You can also review these files for guidelines for modifying an existing custom Tcl interpreter for use as a base application.

The sample application, described in "The Sample Application" on page 109, uses these files to create the interpreters for the sample applications. You can review the makefile in the *demos/sampleApp* subdirectory for guidelines for creating your own makefiles.

Table 14 Base Application Template Files

Interpreter Template File	Description
lib/tcl8.2/proTclWinMain.c	Default implementation of a Windows Tcl interpreter
lib/tk8.2/proTkWinMain.c	Default implementation of a Windows Tk interpreter
lib/tcl8.2/proTclUnixMain.c	Default implementation of a Unix Tcl interpreter
lib/tk8.2/proTkUnixMain.c	Default implementation of a Unix Tk interpreter

Linking Windows Base Applications

The TclPro Wrapper libraries are available in only static versions. However, you must use different versions of the library depending on whether you are creating a statically- or dynamically-linked base application.

The Windows TclPro Wrapper libraries are located in the *win32-ix86\lib* subdirectory of the TclPro installation directory. Table 15 lists the Windows TclPro Wrapper libraries shipped with TclPro.

The TclPro distribution ships two types of static libraries for creating base applications: a static library compiled with **/MT** and one compiled with **/MD**. These files are in the *win32-ix86/lib* directory.

Table 15 Windows TclPro Wrapper Libraries

Library Name	Description
wrapper10x.lib	TclPro Wrapper library for dynamically-linked base applications (compiled with **/MD**)
wrapper10xd.lib	TclPro Wrapper library for dynamically-linked base applications, debug version (compiled with **/MDd**)
wrapper10s.lib	TclPro Wrapper library for statically-linked base applications (compiled with **/MT**)
wrapper10sd.lib	TclPro Wrapper library for statically-linked base applications, debug version (compiled with **/MTd**)

The convention is used that names of the libraries for use with dynamically-linked base applications end with the letter “x”. Use the “s” libraries to create statically-linked base applications.

If you link against the “x” library, link against *MSVCRT.LIB*; if you link against the “xd” library, link against *MSVCRTD.LIB*. If you link against the “s” library, link against *LIBCMT.LIB*; if you link against the “sd” library, link against *LIBCMTD.LIB*.

Linking Unix Base Applications

On Unix systems, there is only one version of the TclPro Wrapper library, which is named *libwrapper10s.a*. The library is contained in the platform-specific library directory, as shown in Table 9 on page 108. (For example, the Linux library is *linux-ix86/lib/libwrapper10s.a*.)

Modifying the Base Application Default Main Files

The **Pro_WrapTclMain** and **Pro_WrapTkMain** procedures replace the standard **Tcl_Main** and **Tk_Main** procedures in a base application. The source for these procedures are contained in *proWrapTclMain.c* or *proWrapTkMain.c*, which are located respectively in the *lib/tcl8.2* and *lib/tk8.2* subdirectories of the TclPro installation directory.

Unless you have modified the standard **Tcl_Main** or **Tk_Main** procedures, you should not have to make any changes to the *proWrapTclMain.c* or *proWrapTkMain.c* files. If you have modified the standard **Tcl_Main** or **Tk_Main** procedures and need to make similar modifications to **Pro_WrapTclMain** or **Pro_WrapTkMain**, you can simply copy *proWrapTclMain.c* or *proWrapTkMain.c* and modify the copy as needed.

If it is not feasible for you to make the modifications to a copy of *proWrapTclMain.c* or *proWrapTkMain.c* (for example, the modifications are too extensive), you can change your existing interpreter code as follows:

1) Change the **Tcl_Main** or **Tk_Main** declarations and definitions to **Pro_WrapTclMain** or **Pro_WrapTkMain**.
2) In your application initialization, determine if the current application is wrapped, and if so, perform some additional processing on the command line arguments. For example, for **Pro_WrapTclMain**:

```
if (Pro_WrapIsWrapped(Tcl_GetNameOfExecutable(),
    &wrappedStartupFileName, &wrappedArgs)) {
    if (wrappedStartupFileName != NULL) {
        fileName = wrappedStartupFileName;
    }
    if (wrappedArgs != NULL) {
        saveArgc = argc;
        saveArgv = argv;
        Pro_WrapPrependArgs(wrappedArgs, saveArgc, saveArgv, \
                            &argc, &argv);
        newArgv = argv;
    }
}
if (wrappedStartupFileName == NULL) {
    if ((argc > 1) && (argv[1][0] != '-')) {
        fileName = argv[1];
        argv0 = fileName;
        argc--;
        argv++;
    }
}
```

This code segment sets things up so that, if the application is wrapped, the wrapped start-up script file (if it was set with the **-startup** option of **prowrap**) is sourced after all other initialization. Also, if additional arguments were supplied during the wrapping process (with the **-arguments** flag), they are inserted between *argv[0]* and *argv[1]*. If the application is not wrapped, the code behaves exactly like **Tcl_Main** and **Tk_Main**.

3) Call **Pro_WrapInit** to initialize the Wrapper library:

```
Pro_WrapInit(interp);
```

Note If your application creates multiple interpreters, you need to called **Pro_WrapInit** only once, for the main interpreter.

4) Free the *newArg* variable near the end of your **Pro_WrapTclMain** or **Pro_WrapTkMain** implementation.

```
if (newArgv != NULL) {
    ckfree((char *) newArgv);
    argc = saveArgc;
    argv = saveArgv;
}
```

Appendix A
Scriptics License Server

Scriptics License Server manages Shared Network Licenses for the Scriptics products in use at your site. A Shared Network License can replace several Named User License keys, provide easy TclPro access to a larger number of developers, and eliminate the need for users to manage and install their own license keys. Scriptics License Server also maintains records about the usage of TclPro products for your reference.

How Licensing Works

All Scriptics products require a license to run. Scriptics sells both *Named User Licenses* and *Shared Network Licenses*. A Named User License allows one specific person to use TclPro. Shared Network Licenses allow anyone at your site to use the TclPro applications, as long as the number of concurrent TclPro users doesn't exceed the number of Shared Network Licenses that you purchase.

This section describe how TclPro applications determine which license to use and how the Scriptics License Server manages Shared Network Licenses.

How TclPro Applications Obtain Licenses

When a user runs a TclPro application, it attempts to obtain a license. The procedure it follows depends on whether the user has installed a local copy of TclPro or is using a shared copy from a server.

If the user has a local copy of TclPro, he or she must have entered license information, either during installation or after wards by running the TclPro License Manager, as described in "Entering TclPro License Information" on page 3. In this case, the application attempts to obtain a license in the following order of precedence:

1) If the user entered a valid permanent Named User License, the application uses that license.

2) If the user entered a hostname and port of a Scriptics License Server, the application attempts to obtain a Shared Network License from that server.

If the user is using a shared copy of TclPro from a server, he or she can either use the default Scriptics License Server for that shared installation (which is set by the site administrator when he or she installs that copy of TclPro), or he or she can run the TclPro License Manager to override that default. In this case, the TclPro application attempts to obtain a license in the following order of precedence:

1) If the user ran TclPro License Manager and entered a valid permanent Named User License, the application uses that license.
2) If the user ran TclPro License Manager and entered a hostname and port of a Scriptics License Server, the application attempts to obtain a Shared Network License from that server.
3) Otherwise, the application attempts to obtain a Shared Network License from the default Scriptics License Server for that installation.

How Scriptics License Server Manages Shared Network Licenses

Scriptics License Server allows a maximum number of concurrent users equal to the number of Shared Network Licenses that you purchase. For example, if you have purchased 10 Shared Network Licenses, then up to 10 users can use TclPro applications at the same time. A user is determined by their user account. The same user account on different hosts counts as only one user. A single user using more than one TclPro application at the same time also counts as only one user.

License Overdraft

Scriptics includes a generous “overdraft” policy with Scriptics License Server that allows you to exceed your concurrent user limit in emergency situations. This policy accommodates occasions where you temporarily need additional licenses before you have had time to purchase them.

When Scriptics License Server receives a request for a license in excess of the number of licenses you have purchased, Scriptics License Server records an “overdraft day.” Multiple overdraft instances on a single day count as only one overdraft day.

For the first 10 overdraft days that occur, Scriptics License Server continues to issue “overdraft licenses,” which allow the TclPro applications causing the overdraft to continue to run. After 10 overdraft days, the Scriptics License Server no longer issues overdraft licenses and strictly enforces the concurrent user limit.

When a TclPro application causes an overdraft, it displays a warning message to the user. Scriptics License Server also notifies the site administrator by email. The Scriptics License Server daily and weekly reports include the number of licenses in use and the number of overdraft occurrences. See “Viewing Reports” on page 127 for more information on reporting.

Scriptics License Server Installation

Scriptics License Server runs on Unix systems only. You should install Scriptics License Server on a reliable server that is accessible by all TclPro users. You don’t have to install Scriptics License Server and TclPro on the same system.

Typically, the server starts Scriptics License Server automatically using a standard *init.d* script, which is created automatically during installation. You rarely should need to start or stop Scriptics License Server manually.

Installing the Scriptics License Server Software

You can install Scriptics License Server from either the TclPro CD or the Unix installation download available from the Scriptics web site (*http://www.scriptics.com/tclpro*). Run *setup.sh* and select the Scriptics License Server option.

Important Log in as the *root* user to install Scriptics License Server.

The installation program prompts you for a port number for the Scriptics License Server. Although you can select any free port on your system, Scriptics recommends that you select the default value of port 2577.

Setting the Initial Configuration

After installing Scriptics License Server, you must configure it through its Web browser interface. To display the Scriptics License Server Web interface, launch a Web browser and open the following URL:

```
http://hostname:port/
```

hostname is the hostname of the system running Scriptics License Sever. *port* is the port number you assigned during installation.

The first time you connect to the Scriptics License Server Web interface, it displays the Set Initial Configuration page that prompts for initial configuration information:

Your Company Name

Your company name appears on the main page of the Web interface. This name is also displayed by programs that get licenses from this server.

Administrative Name and Password

Access to the administrative pages are password protected. On the Set Initial Configuration page you choose the name and password for the first administrator account. You can define other name/password pairs or change existing one from the Change Passwords page (*/admin/password.tml*).

Email Contact Address

Scriptics License Server send email messages to the site administrator that contain usage reports as well as problem notifications. You can add more email addresses and tune what events trigger email later using the Email page (*/admin/email.tml*).

OK to Email Scriptics

Scriptics License Server send email messages to Scriptics for problem notification. You change this setting later using the Email page (*/admin/email.tml*).

After you configure Scriptics License Server, opening *http://<hostname>:<port>* displays the Scriptics License Server Home page. From that page, you can administer server settings, manage license keys, and generate reports. See "Scriptics License Server Administration" on page 126 for more information.

Note You can reset Scriptics License Server and delete all configuration information except the license keys you have installed (but including administrator names and passwords) by executing **prolserver -reset**. After resetting Scriptics License Server, it displays the Set Initial Configuration page the next time you open its Web interface.

Scriptics License Server Installed Files

The installation program installs the following files (*<installDir>* is the installation directory you specify during installation):

/etc/init.d/prolserver

The shell script that starts Scriptics License Server when the system boots. You can use run this script with the **start** or **stop** argument to start or stop Scriptics License Server manually. The exact location of this file depends on your operating system (for example, */etc/init.d*,

/etc/rc.d/init.d, or */sbin/init.d*). The peer directories *rc2.d* and *rc0.d* contain symbolic links to this file, which your system uses to start Scriptics License Server during boot and halt it during shutdown.

<installDir>/prolserver.boot

This script is created only if you do not install Scriptics License Server as *root*. This is a copy of the script that needs to be installed under */etc/init.d* in order to automatically launch Scriptics License Server when the system boots. The exact location depends on the operating system.

<installDir>/prolserver

The Scriptics License Server application program.

<installDir>/prolserver.conf

The main configuration file. This file stores the installation settings for Scriptics License Server. Consult the **prolserver** man page for information about the setting for this file. Changes to this file take place when you restart Scriptics License Server.

<installDir>/prolserver.state

This is a state checkpoint file. This file is tamper-resistent so you should not edit this file. Doing so causes you to lose state about active licenses, administrator passwords, and notification email addresses.

<installDir>/prolserver.pid

This file records the process ID of Scriptics License Server so it can be shutdown.

<installDir>/htdocs/

A directory containing the files implementing the Scriptics License Server Web interface.

/var/log/prolserver.<port>.log

This is the log file for the server. *<port>* is the port number assigned to this Scriptics License Server. This file keeps weekly, daily, and current usage statistics. It is compacted automatically so it should not get too large. Each day at midnight the records about the current usage are collected into one daily record, and each week records about daily usage are collected into one weekly record. You can change the location of this file by editing the *prolserver.conf* file. Consult the **prolserver** man page for details.

/etc/INSTALL.LOG

A log of the Scriptics License Server installation.

Scriptics License Server Administration

You manage Scriptics License Server using a Web browser interface. To display the Scriptics License Server home page, launch a Web browser and open the following URL:

```
http://hostname:port/
```

hostname is the hostname of the system running Scriptics License Sever. *port* is the port number you assigned during installation.

The Scriptics License Server Web interface provides several pages for administering server settings, managing license keys, and generating reports. Access to administrative pages is password protected using the Basic Authentication scheme supported by all browsers. When you initially configure your Scriptics License Server, you specify the name and password for the first administrator account. You can define other name/password pairs or change existing ones from the Change Passwords page (*/admin/password.tml*).

Each page contains documentation describing the information displayed and the actions you can perform. Therefore, this guide provides only an overview of the Web interface. Consult the Web interface for more detailed information.

Managing Licenses

Shared Network Licenses are distributed as encoded keys. You can add, upgrade, and delete Shared Network License keys from the Manage Licenses page (*/admin/license.tml*). This page also displays the license keys currently installed.

License keys are specific to a TclPro release (for example, 1.2). When new releases appear, Update Service customers can upgrade their keys automatically. The “Upgrade Key” buttons on the Manage Licences page contact Scriptics and verify your eligibility for the upgrade. If eligible, a new license key is returned and added to your system automatically.

Revoking Licenses

In some circumstances you may need to revoke a license in use by one user so that another user can obtain the license. For example, a user may have gone on vacation while leaving TclPro Debugger running. The Revoke Active Licenses page (*/admin/revoke.tml*) allows you to revoke individual licenses in use.

Changing Email Notifications

The license server can generate email notifications when various events occur. The Email page (*/admin/email.tml*) allows you to specify which users get email in response to which kind of events.

Setting Date Formats

The Date Format page (*/admin/dateformat.tml*) allows you to specify the date format to use when Scriptics License Server generates reports.

Viewing Reports

Scriptics License Server generates a variety of reports about usage of TclPro tools. There is a daily view and a weekly view. Both views list the TclPro applications and the number of times they have been used each day (or week). The reports also list system events such as Overdraft conditions and License Denied. Either of these events indicate that your site may not be configured with enough Shared Network Licenses.

All reports are available from the License Reports page (*/reports/index.tml*).

Appendix B
TclPro Checker Messages

Table 16 lists the messages that TclPro Checker can produce.

Table 16 TclPro Checker Messages

Message ID	Message Type	Explanation
argAfterArgs	Error	Argument specified after “args”
argsNotDefault	Error	“args” cannot be defaulted
badBoolean	Error	Invalid Boolean value
badByteNum	Error	Invalid number, should be between 0 and 255
badColorFormat	Error	Invalid color name
badColormap	Error	Invalid colormap “*colormap*”: must be “new” or a window name
badCursor	Error	Invalid cursor spec
badEvent	Error	Invalid event type or keysym
badFloat	Error	Invalid floating-point value
badGeometry	Error	Invalid geometry specifier
badGridMaster	Error	Cannot determine master window
badGridRel	Error	Must specify window before shortcut
badIndex	Error	Invalid index: should be integer or “end”
badInt	Error	Invalid integer
badKey	Error	Invalid keyword “*key*” must be: *words*
badLevel	Error	Invalid level

Table 16 TclPro Checker Messages *(Continued)*

Message ID	Message Type	Explanation
badLIndex	Error	Invalid infants: should be integer, “len” or “end”
badList	Error	Invalid list: *error-info*
badMemberName	Error	Missing class specifier for body declaration
badMode	Error	Access mode must include either RDONLY, WRONLY, or RDWR
badOption	Error	Invalid option “*option*” must be: *options*
badPalette	Error	Invalid palette spec
badPixel	Error	Invalid pixel value
badPriority	Error	Invalid priority keyword or value
badProfileOpt	Error	Option “*option*” not valid when turning off profiling
badResource	Error	Invalid resource name
badScreen	Error	Invalid screen value
badSticky	Error	Invalid stickiness value: should be one or more of nswe
badSwitch	Error	Invalid switch: “*switch*”
badTab	Error	Invalid tab list
badTabJust	Error	Invalid tab justification “t*ab-item*”: must be left right center or numeric
badTlibFile	Error	The filename must have a “.tlib” suffix
badTraceOp	Error	Bad operation *operation* should be one or more of rwu
badVersion	Error	Invalid version number
badVirtual	Error	Virtual event is badly formed
badVisual	Error	Invalid visual
badVisualDepth	Error	Invalid visual depth
badWholeNum	Error	Bad value “*value*”: must be a non-negative integer
classNumArgs	Error	Wrong # args for class constructor: *className*
classOnly	Error	Command “*command*” only defined in class body

Table 16 TclPro Checker Messages *(Continued)*

Message ID	Message Type	Explanation
errBadBrktExp	Error	The bracket expression is missing a close bracket
mismatchOptions	Error	The specified options cannot be used in tandem
noEvent	Error	No events specified in binding
noExpr	Error	Missing an expression
noScript	Error	Missing a script after "*control*"
noSwitchArg	Error	Missing argument for *switch* switch
noVirtual	Error	Virtual event not allowed in definition of another virtual event
nonDefAfterDef	Error	Non-default arg specified after default
nonPortBitmap	Non-Portable Warning	Use of non-portable bitmap
nonPortChannel	Non-Portable Warning	Use of a non-portable file descriptor, use "file" instead
nonPortCmd	Non-Portable Warning	Non-portable command
nonPortColor	Non-Portable Warning	Non-portable color name
nonPortCursor	Non-Portable Warning	Non-portable cursor usage
nonPortFile	Non-Portable Warning	Use of non-portable file name, use "file join"
nonPortKeysym	Non-Portable Warning	Use of non-portable keysym
nonPortOption	Non-Portable Warning	Use of non-portable option
nonPortVar	Non-Portable Warning	Use of non-portable variable
nsOnly	Error	Command "*command*" only defined in namespace body
nsOrClassOnly	Error	Command "*command*" only defined in class or namespace body
numArgs	Error	Wrong # args
obsoleteCmd	Error	Deprecated usage, use "*command*" instead
optionRequired	Error	Expected *option1*, got "*option2*"
parse	Error	Parse error: *error-info*
procNumArgs	Error	Wrong # args for the user-defined proc: *procName*.

Table 16 TclPro Checker Messages *(Continued)*

Message ID	Message Type	Explanation
procOutScope	Error	Proc only defined in class *className*
procProtected	Error	Calling *protectionLevel* proc: *procName*
serverAndPort	Error	Option -myport is not valid for server sockets
socketAsync	Error	Cannot use -server option and -async option
socketServer	Error	Cannot use -async option for server sockets
tooManyFieldArg	Error	Too many fields in argument specifier
warnAmbiguous	Usage Warning	Ambiguous switch, use *delimiter* to avoid conflicts
warnDeprecated	Upgrade Warning	Deprecated usage, use "*command*" instead
warnEscapeCharacter	Upgrade Warning	"\<*char*>" is a valid escape sequence in later versions of Tcl.
warnExportPat	Warning	Export patterns should not be qualified
warnExpr	Performance Warning	Use curly braces to avoid double substitution
warnExtraClose	Usage Warning	Unmatched closing character
warnIfKeyword	Warning	Deprecated usage, use else or elseif
warnNamespacePat	Warning	glob chars in wrong portion of pattern
warnNotSpecial	Upgrade Warning	"\<*char*>" has no meaning. Did you mean "\\<*char*>" or "<*char*>"?
warnPattern	Warning	Possible unexpected substitution in pattern
warnQuoteChar	Upgrade Warning	"\" in bracket expressions are treated as quotes
warnRedefine	Usage Warning	*userProc1* redefines *userProc2* in file *fileName* on line *lineNum*
warnReserved	Upgrade Warning	Keyword is reserved for use in *version*
warnUndefProc	Warning	The procedure was called but was never defined
warnUnsupported	Error	Unsupported command, option or variable: use *command*
warnVarRef	Warning	Variable reference used where variable name expected

Table 16 TclPro Checker Messages *(Continued)*

Message ID	Message Type	Explanation
warnY2K	Warning	%y generates a year without a century. consider using %Y to avoid Y2K errors
winAlpha	Error	Window name cannot begin with a capital letter
winBeginDot	Error	Window name must begin with “.”
winNotNull	Error	Window name cannot be an empty string

TclPro Checker Message Descriptions

This section provides detailed descriptions of the TclPro Checker messages.

argAfterArgs

Message String:

Argument specified after “args”

Category: Error

Explanation:

An argument has been specified after the **args** keyword in a procedure argument list. The **args** argument is treated like a normal parameter and does not collect the remaining parameters into a single list.

argsNotDefault

Message String:

“args” cannot be defaulted

Category: Error

Explanation:

The **args** keyword cannot be initialized to contain a default value. Although the Tcl interpreter does not complain about this usage, the default value is ignored.

badBoolean

Message String:

Invalid Boolean value

Category: Error

Explanation:

The command expects the string to specify a Boolean value. The string can be “1”, “0”, “true”, “false”, “yes”, “no”, “on”, or “off” in any unique abbreviation and case.

badByteNum

Message String:

Invalid number, should be between 0 and 255

Category: Error

Explanation:

The type should be a integer between 0 and 255.

badColorFormat

Message String:

Invalid color name

Category: Error

Explanation:

The command expects the string to specify a color value. The string can be any of the following forms:

colorname

#RGB

#RRGGBB

#RRRGGGBBB

#RRRRGGGGBBBB

colorname can be any of the valid textual names for a color defined in the server's color database file, such as “red” or “PeachPuff”. If the color name is not a Tcl defined color, a warning is flagged stating that the color may not be portable across all platforms; see nonPortColor. The RGB characters represent hexadecimal digits that specify the red, green, and blue intensities of the color.

badColormap

Message String:

Invalid colormap “*colormap*”: must be “new” or a window name

Category: Error

Explanation:

The command expects the string to specify a colormap to use. If the string is “new”, a new colormap is created. Otherwise, the string should be a valid window path name.

badCursor

Message String:

Invalid cursor spec

Category: Error

Explanation:

The command expects the string to specify a cursor to use. The string can take any of the following forms:

“”

name

name fgColor

@sourceFile fgColor

name fgColor bgColor

@sourceFile maskFile fgColor bgColor

If the *name* form is used, and the name of the cursor is not defined on all platforms, a warning is flagged stating that the cursor is not portable; see nonPortCursor. None of the forms that specify a color or multiple files are portable across all systems; they are flagged as being non-portable; see nonPortCmd.

badEvent

Message String:

Invalid event type or keysym

Category: Error

Explanation:

The command expects the string to specify an event type. If the string is not composed of a valid event and one or more related modifiers, an error is reported.

badFloat

Message String:

Invalid floating-point value

Category: Error

Explanation:

The command expects the string to consist of a floating-point number, which is: white space; a sign; a sequence of digits; a decimal point; a sequence of digits; the letter “e”; and a signed decimal exponent. Any of the fields may be omitted, except that the digits either before or after the decimal point must be present and if the “e” is present then it must be followed by the exponent number.

badGeometry

Message String:

Invalid geometry specifier

Category: Error

Explanation:

The command expects the string to specify a geometry value. The string must have one of the following forms:

*W*x*H*

±*X*x±*Y*

*W*x*H*±*X*x±*Y*

where the width (*W*) and height (*H*) values are positive integers, and the X (*X*) and Y (*Y*) coordinates are positive or negative integers.

badGridMaster

Message String:

Cannot determine master window

Category: Error

Explanation:

The **grid** command flags an error if a valid window name was never specified in the command.

badGridRel

Message String:

Must specify window before shortcut

Category: Error

Explanation:

When using the relative placement shortcuts in the **grid** command (that is, "-", "x", or "^") an error is reported if the span column shortcut ("-") is used immediately after one of the other shortcuts.

badIndex

Message String:

Invalid index: should be integer or **end**

Category: Error

Explanation:

The command expects the string to specify an index value. The string can be an integer or "end".

badInt

Message String:

Invalid integer

Category: Error

Explanation:

The command expects the string to specify an integer value. The string can be optionally signed and optionally preceded by white space. If the first two characters of the string are "0x" then string is expected to be in hexadecimal form; if the first character of string is "0" then the string is expected to be in octal form; otherwise, the string is expected to be in decimal form.

badKey

Message String:

Invalid keyword: "*key*" must be: *options*

Category: Error

Explanation:

The command expects the *key* string to be a key that matches one of the strings in the *options* list.

badLevel

Message String:

Invalid level

Category: Error

Explanation:

The command expects the string to be an integer or a “#” character followed by an integer.

badLIndex

Message String:

Invalid index: should be integer, “len” or “end”

Category: Error

Explanation:

The command expects the string to specify an index value. The string can be an integer, “len”, or “end”.

badList

Message String:

Invalid list: *error-info*

Category: Error

Explanation:

The command expects the string to be a valid Tcl list. The reason the string is not a valid Tcl list is displayed in the message associated with the error.

badMemberName

Message String:

Missing class specifier for body declaration

Category: Error

Explanation:

An [incr Tcl] member name was not correctly qualified. When defining the body for a class procedure, class method, or class variable, it is necessary to reference the procedure or variable with the fully qualified name.

badMode

Message String:

Access mode must include either RDONLY, WRONLY, or RDWR

Category: Error

Explanation:

When specifying access modes for a Tcl channel, at least one of the three read-write access modes (RDONLY, WRONLY, or RDWR) must be specified with optional modifiers (APPEND, CREAT, EXCL, NOCTTY, NONBLOCK or TRUNC).

badOption

Message String:

Invalid option "*option*" must be: *options*

Category: Error

Explanation:

The command expects the *option* string to be an option that matches one of the strings in *options*.

badPalette

Message String:

Invalid palette spec

Category: Error

Explanation:

The command expects the string to be a valid palette specification. The palette string may be either a single decimal number, specifying the number of shades of gray to use, or three decimal numbers separated by slashes ("/"), specifying the number of shades of red, green and blue to use, respectively.

badPixel

Message String:

Invalid pixel value

Category: Error

Explanation:

The command expects the string to specify a pixel value. The string must be an integer pixel or floating-point millimeter, optionally followed by one of the following characters: “c”, “i”, “m”, or “p”.

badPriority

Message String:

Invalid priority keyword or value

Category: Error

Explanation:

The command expects the string to specify a priority value. The string must contain one of the following values: “widgetDefault”, “startupFile”, “userDefault”, “interactive”, or an integer between 0 and 100.

badProfileOpt

Message String:

Option *option* not valid when turning off profiling

Category: Error

Explanation:

Using the TclX profiling tools, *option* is not valid. Most likely the option is valid only when turning on profiling.

badResource

Message String:

Invalid resource name

Category: Error

Explanation:

The command expects the string to specify a resource value. If the string length is not four characters, an error is flagged.

badScreen

Message String:

Invalid screen value

Category: Error

Explanation:

The command expects the string to specify a screen value. The string must have the following form:

?*name*?:*display*?.*screen*?

where *name* is any string and *display* and *screen* are integers.

badSticky

Message String:

Invalid stickiness value: should be one or more of nswe

Category: Error

Explanation:

The **grid** command expects the string to specify valid sticky coordinates. The string can contain any combination of the following characters: “n”, “s”, “e”. or “w”.

badSwitch

Message String:

Invalid switch: “*switch*”

Category: Error

Explanation:

The command expects the string to be a switch that matches one of the strings in list of switch options.

badTab

Message String:

Invalid tab list

Category: Error

Explanation:

The command expects a list of strings that define tab stops. If the Tcl interpreter cannot parse the list, an error is flagged. The tab list must consist of a list of screen distances giving the positions of the tab stops.

Each position can optionally be followed in the next list element by one of the keywords “left”, “right”, “center”, or “numeric”, which specifies how to justify text relative to the tab stop.

badTabJust

Message String:

Invalid tab justification “*tab-item*”: must be left right center or numeric

Category: Error

Explanation:

The command expects the justification string to be one of the following: “left”, “right”, “center”, or “numeric”.

badTlibFile

Message String:

The filename must have a “.tlib” suffix

Category: Error

Explanation:

The command expected a filename with a *.tlib* suffix. The word should be changed to match the pattern *filename.tlib*.

badTraceOp

Message String:

Invalid operation “*op*”: should be one or more of rwu

Category: Error

Explanation:

The command expects the trace operation string to be one or more of the following characters: “r”, “w”, or “u”.

badVersion

Message String:

Invalid version number

Category: Error

Explanation:

The command expects a list of strings that specifies a package version. A valid package version string is any number of integers separated by periods (“.”), for example, “1.2.3”.

badVirtual

Message String:

Virtual event is badly formed

Category: Error

Explanation

The command expects the string to specify a virtual event. The string must have the following form:

<<*word*>>

where *word* is any non-empty string.

badVisual

Message String:

Invalid visual

Category: Error

Explanation:

The command expects the string to specify a visual. The string can have the following form:

class depth

default

windowName

number

best ?*depth*?

The *class* string must be one of “directcolor”, “grayscale”, “pseudocolor”, “staticcolor”, “staticgray”, or “truecolor”, or any unique abbreviation. The *depth* value must be a valid integer.

badVisualDepth

Message String:

Invalid visual depth

Category: Error

Explanation:

If the depth specified by a visual string is not a valid integer, then this error is flagged.

badWholeNum

Message String:

Invalid value "*value*": must be a non-negative integer

Category: Error

Explanation:

The command expects the string to specify a whole value. The string can be any non-negative integer.

classNumArgs

Message String:

Wrong # args for class constructor: *className*.

Category: Error

Explanation:

The wrong number of arguments are being used to instantiate the [incr Tcl] class *className*. Compare the number of arguments used to instantiate the class to the number of arguments in the constructor defined by *className*.

classOnly

Message String:

Command "*command*" only defined in class body

Category: Error

Explanation:

The specified command is only valid in the context of an [incr Tcl] class body.

errBadBrktExp

Message String:

The bracket expression is missing a close bracket

Category: Error

Explanation:

The bracket expression is missing a close bracket. Common errors of this type are caused when the closing bracket is interpreted as a character to match on. For example [] and [^] will generate this error because the close bracket is interpreted as a character to match, or not match, respectively. The correct expressions would be: []] and [^]].

mismatchOptions

Message String:

The specified options cannot be used in tandem

Category: Error

Explanation:

Two or more options were specified that cannot be used at the same time. The command should be re-written to use only one of the switches. This commonly occurs when an overloaded command performs completely different operations based on the switches.

noEvent

Message String:

No events specified in binding

Category: Error

Explanation:

The command expects an event but could not find one while parsing the command line.

noExpr

Message String:

Missing an expression

Category: Error

Explanation:

Similar to the numArgs message. TclPro Checker flags this error message when an expression is missing in an **if** statement.

noScript

Message String:

Missing a script after *control*

Category: Error

Explanation:

Similar to the numArgs message. TclPro Checker flags this error message when a script is missing in an **if** statement.

noSwitchArg

Message String:

Missing argument for *switch* switch

Category: Error

Explanation:

The command was called with a switch that expected an argument. If no argument was given for the switch, this error is flagged.

noVirtual

Message String:

Virtual event not allowed in definition of another virtual event

Category: Error

Explanation:

Virtual events are not allowed in event sequences. If a virtual event (any event that begins with “<<” and ends with “>>”) is found, then this message is flagged.

nonDefAfterDef

Message String:

Non-default arg specified after default

Category: Error

Explanation:

A non-defaulted argument has been specified after a defaulted argument in a procedure argument list. Although the Tcl interpreter does not complain about this usage, the default values are ignored.

nonPortBitmap

Message String:
Use of non-portable bitmap

Category: Non-Portable Warning

Explanation:
A bitmap was specified that is not supported on all platforms.

nonPortChannel

Message String:
Use of non-portable file descriptor, use “file” instead

Category: Non-Portable Warning

Explanation:
A channel was specified that is not supported on all platforms. In most cases, this is when “file0”, “file1”, or “file2” is used instead of “stdin”, “stdout”, or “stderr”.

nonPortCmd

Message String:
Use of non-portable command

Category: Non-Portable Warning

Explanation:
A command was specified that is not supported on all platforms.

nonPortColor

Message String:
Non-portable color name

Category: Non-Portable Warning

Explanation:
A color was specified that is not supported on all platforms.

nonPortCursor

Message String:
Non-portable cursor usage

Category: Non-Portable Warning

Explanation:
A cursor was specified that is not supported on all platforms.

nonPortFile

Message String:
Use of non-portable file name, use file join

Category: Non-Portable Warning

Explanation:
A file name was specified that is not supported on all platforms. This warning is flagged, then the string is a combination of words, variables, or commands separated by system-specific file separators (for example, "$dir\$file"). Use the **file join** command to add the system-specific file separators (for example, "[file join $dir $file]").

nonPortKeysym

Message String:
Use of non-portable keysym

Category: Non-Portable Warning

Explanation:
A keysym was specified that is not supported on all platforms.

nonPortOption

Message String:
Use of non-portable option

Category: Non-Portable Warning

Explanation:
An option was specified that is not supported on all platforms. Generally, the option has no effect on the systems that do not support this option.

nonPortVar

Message String:
Use of non-portable variable

Category: Non-Portable Warning

Explanation:

A variable was used that is not supported on all platforms. In most cases, this is when the *tcl_precision* variable is used.

nsOnly

Message String:

Command "*command*" only defined in namespace body

Category: Error

Explanation:

The specified command is only valid in the context of an [incr Tcl] namespace body.

nsOrClassOnly

Message String:

Command *command* only defined in class or namespace body

Category: Error

Explanation:

The specified command is only valid in the context of an [incr Tcl] class or namespace body.

numArgs

Message String:

Wrong # args

Category: Error

Explanation:

An incorrect number of arguments were specified for a command. Due to the dynamic nature of Tcl, this error might be flagged unnecessarily. For example, if the command is called within an **eval** body with variables that will expand to be multiple arguments. TclPro Checker sees only the one argument, but this may expand to match the required number of arguments when the command is evaluated.

obsoleteCmd

Message String:

Deprecated usage, use "*command*" instead

Category: Error

Explanation:

The specified command, option or variable does not exist and is no longer supported in the version of the system you are checking. Use the suggested alternative command, option, or variable to upgrade the script.

optionRequired

Message String:

Expected *option1*, got "*option2*"

Category: Error

Explanation:

A specific option was expected, but the following option was found.

parse

Message String:

Parse error: *error-info*

Category: Error

Explanation:

TclPro Checker could not parse the script completely due to a parsing error. The reason for the parsing error is displayed in the message associated with the error.

procNumArgs

Message String:

Wrong # args for user-defined proc: *procName*

Category: Error

Explanation:

You are using the wrong number of arguments to call the Tcl procedure *procName*. Compare the number of arguments used to call the procedure to the number of arguments in the definition of *procName*.

procOutScope

Message String:

Proc only defined in class *className*

Category: Error

Explanation:

An [incr Tcl] class procedure is being called from the wrong scope, or is improperly qualified. This commonly occurs when calling inherited procedures.

procProtected

Message String:

Calling *protectionLevel* proc: *procName*

Category: Error

Explanation:

You are calling an inaccessible procedure with a protection level of *protectionLevel*. This error is flagged when the procedure being called does not have permission to call this procedure.

serverAndPort

Message String:

Option -myport is not valid for server sockets

Category: Error

Explanation:

The socket command specified the **-server** option and the **-myport** option on the same command line. These are conflicting options and cannot be used together.

socketAsync

Message String:

Cannot use -server option and -async option

Category: Error

Explanation:

The socket command specified the **-server** option and the **-async** option on the same command line. These are conflicting options and cannot be used together.

socketServer

Message String:

Cannot use -async option for server sockets

Category: Error

Explanation:

The socket command specified the **-async** option and the **-server** option on the same command line. These are conflicting options and cannot be used together.

tooManyFieldArg

Message String:

Too many fields in argument specifier

Category: Error

Explanation:

A defaulted procedure argument has been specified with multiple values. An argument can have only one default value. If the value is to be a list, quotes or curly braces must be used.

warnAmbiguous

Message String:

Ambiguous switch, use *delimiter* to avoid conflicts

Category: Usage Warning

Explanation:

The word being checked starts with a "-" but does not match any of the known switches. Use *delimiter* to explicitly declare the end of the switch pattern.

warnDeprecated

Message String:

Deprecated usage, use "*command*" instead

Category: Upgrade Warning

Explanation:

The specified command, option or variable does not exist and is no longer supported in the version of the system you are checking. Use the suggested alternative command, option or variable to upgrade the script.

warnEscapeCharacter

Message String:

"\<*char*>" is a valid escape sequence in later versions of Tcl.

Category Upgrade Warning

Explanation:

The new regular expression package introduced in Tcl 8.1 added many new special character sequences, called "escape characters." When upgrading to 8.1 or later, the escape characters will change the semantics of the expression. To maintain the behavior of previous versions of Tcl, add another backslash before the character. (This warning is displayed only if you specify the **-use** option with Tcl 8.0 or earlier.)

warnExportPat

Message String:

Export patterns should not be qualified

Category: Warning

Explanation:

Each export pattern can contain **glob**-style special characters, but it must not include any namespace qualifiers. That is, the pattern can only specify commands in the current (exporting) namespace.

warnExpr

Message String:

Use curly braces to avoid double substitution

Category: Performance Warning

Explanation:

The **expr** command performs two levels of substitution on all expressions that are not inside curly braces. To avoid the second substitution, and to improve the performance of the command, place the expression inside curly braces.

Note There are cases where the second level of substitution is required and this warning will not apply. TclPro Checker does not discern between these cases.

warnExtraClose

Message String:

Unmatched closing character

Category: Usage Warning

Explanation:

A close bracket or close brace without a matching open bracket or open brace was detected. This frequently indicates an error introduced when a subcommand or script is deleted without deleting the final close brace or bracket.

warnIfKeyword

Message String:

Deprecated usage, use else or elseif

Category: Warning

Explanation

When using the **if** command, it is legal to omit the **else** and **elseif** keywords. However, omission of these keywords tends to produce error-prone code; thus, a warning is flagged.

warnNamespacePat

Message String:

glob chars in wrong portion of pattern

Category: Warning

Explanation

Each **namespace** pattern is qualified with the name of an exporting **namespace** and may have **glob**-style special characters in the command name at the end of the qualified name. The warning is flagged if **glob** characters appears in a **namespace** name.

warnNotSpecial

Message String:

"\<*char*>" has no meaning. Did you mean "\\<*char*>" or "<*char*>"?

Category: Upgrade Warning

Explanation:

The backslash character is used to quote special characters in an expression so their literal value can be used. The character following the backslash in this expression has no affect on the character. Consider simplifying the expression.

warnPattern

Message String:

Possible unexpected substitution in pattern

Category: Warning

Explanation:

Glob patterns use brackets to specify a list of characters to match. If brackets are used and the word is not properly quoted, Tcl will interpret this as a subcommand to be evaluated, rather than a pattern. This warning is flagged to avoid possible usage errors of this nature.

warnQuoteChar

Message String:

"\" in bracket expressions are treated as quotes

Category: Upgrade Warning

Explanation:

The new regular expression package introduced in Tcl 8.1 changed the semantics of the backslash character inside of bracket expressions. Previously they were treated as literal characters. Now they are treated as a quote character. To maintain the behavior of previous versions of

Tcl, add another backslash before the existing backslash (for example, [*-\] becomes [*-\\]). (This warning is displayed only if you specify the **-use** option with Tcl 8.0 or earlier.)

warnRedefine

Message String:

userProc1 redefines *userProc2* in file *fileName* on line *lineNum*

Category: Usage Warning

Explanation

A procedure or class is being defined, imported, inherited, or renamed into a scope where a procedure or class of the same name already exists.

warnReserved

Message String:

Keyword is reserved for use in *version*

Category: Upgrade Warning

Explanation

When checking scripts using older versions of Tcl, Tk or [incr Tcl], this warning is flagged if a command is used that does not exist in the systems that you are checking against, but does exist in later versions. This warning helps to prevent scripts from defining commands that will eventually collide with later versions.

warnUndefProc

Message String:

The procedure was called but was never defined

Category: Warning

Explanation:

The procedure was not defined in any of the files that were specified on the command line of the current invocation of TclPro Checker. The procedure may get defined dynamically or in a file that was not specified on the TclPro Checker command line. This warning is triggered only for the first use of the undefined procedure in the files being checked.

warnUnsupported

Message String:

Unsupported command, option or variable: use *command*

Category: Error

Explanation:

The specified command, option or variable still exists but is no longer supported. Use the suggested alternative command, option, or variable to upgrade the script.

warnVarRef

Message String:

Variable reference used where variable name expected

Category: Warning

Explanation

Some commands expect a variable name for an argument, for example, **incr**. If the argument is a variable reference, this warning is flagged to report possible usage errors.

warnY2K

Message String:

%y generates a year without a century. consider using %Y to avoid Y2K errors.

Category: Warning

Explanation:

To avoid possible Y2K errors, use the “%Y” field descriptor to generate years with centuries (for example, “1999” instead of “99”).

winAlpha

Message String:

Window name cannot begin with a capital letter

Category: Error

Explanation

The window name for any Tcl widget cannot begin with a capital letter.

winBeginDot

Message String:

Window name must begin with “.”

Category: Error

Explanation

The path name for any Tcl widget must begin with a period (“.”)

winNotNull

Message String:

Window name cannot be an empty string

Category: Error

Explanation

A window name or path cannot be an empty string.

Index

D

E

F

G

P

R

S

T

U

V

W