Agilent Technologies Enterprise Link

Developer's Guide

Part No. E2700-90020
For use with Enterprise Link release E.02.30
for Windows NT 4.0

Printed in USA March 2000

Notices

The information contained in this manual is subject to change without notice. Agilent Technologies makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Agilent Technologies shall not be liable for errors contained herein or direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of the material.

TRADEMARKS

Adobe® is a trademark of Adobe Systems Incorporated which may be registered in certain jurisdictions.

HP-UX Release 10.20 and later and HP-UX Release 11.00 and later (in both 32 and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Windows®, MS Windows®, Windows NT® are U.S. registered trademarks of Microsoft Corporation.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Agilent Technologies, Inc.
395 Page Mill Road
Palo Alto, CA
94303-0870, USA

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Enterprise Link Documentation Set

The Enterprise Link documentation set includes four books.

HP-UX Installation Guide

Describes installing Enterprise Link on the HP-UX operating system. You can view this book online using Adobe®Acrobat Reader.

Windows NT Installation Guide

Describes installing Enterprise Link on the Windows NT operating system. You can view this book online using AdobeAcrobat Reader.

User's Guide

Describes Enterprise Link and provides step-by-step procedures for using Enterprise Link. You can view this guide online using Adobe Acrobat Reader, or order a preprinted copy.

Developer's Guide

Describes how to extend Enterprise Link to communicate with other systems. You can view this guide online using Adobe Acrobat Reader, or order a preprinted copy.

In This Book

This book describes how to extend the Enterprise Link product to communicate with new kinds of applications.

- Chapter 1 presents an overview of Enterprise Link's data server and configuration tool. In addition, this chapter provides conceptual information needed to extend this product.
- Chapter 2 describes how to build a version of the Enterprise Link Tcl interpreter, as well as the behavior of the build script.
- Chapter 3 describes how to develop a communication object for the data server.
- Chapter 4 describes how to develop a communication object for the configuration tool, including developing the interface class.
- Chapter 5 describes how to develop the optional system-specific Access window for the configuration tool.
- Chapter 6 describes how to develop the optional system-specific Trigger panel for the configuration tool.
- Chapter 7 documents the communication classes for the data server, configuration tool, and spooler.
- Chapter 8 documents utilities that simplify developing the communication objects for the data server and configuration tool.

4 Developing the Configuration Tool Communication Object

5 Developing an Access Window

6 Developing a Trigger Panel

7 Class Reference

8 Utility Reference

Index

About this Edition

Need Assistance?

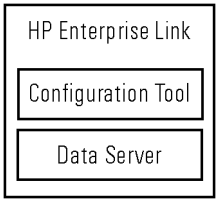
1

Concepts

Concepts

Agilent Technolgies' Enterprise Link can connect two distinct software systems. Enterprise Link comes ready-configured to link RTAP and SAP; however, you can extend Enterprise Link's capabilities to include other systems. This book describes how to extend Enterprise Link to communicate with new systems. This chapter provides conceptual information you should read before extending Enterprise Link.

Enterprise Link includes a graphical user interface component (the configuration tool) and a run-time component (the data server).



To extend Enterprise Link to communicate with new systems, you have to write new communication classes for the data server (elserver) and configuration tool (elconfig). This process usually involves three activities:

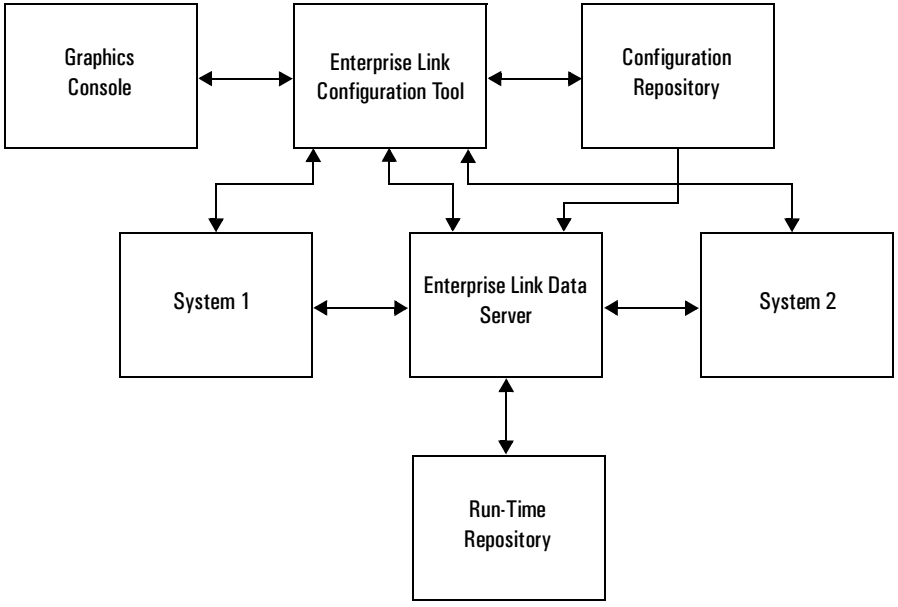
1. Extend the Tcl interpreter, usually with C code, so that it can manipulate the required parts of the new system's API (see "Tcl/C API Interface" on page 1-18).
2. Write a new communication class for the data server in Tcl, deriving the class from the **elCommClass** base class. The base class is located in $ELROOT/lib/elink/elserver_link.tcl on HP-UX systems, and in %ELROOT%\lib\elink\elserver_link.tcl on Windows NT systems (see chapter 3, "Developing the Data Server Communication Object").
3. Write a new communication class for the configuration tool (see chapter 4, "Developing the Configuration Tool Communication Object").

Caution

Much of Enterprise Link is shipped in source-code form as Tcl scripts. The contents of these Tcl scripts may change radically from one release of Enterprise Link to the next. Base your code on the functionality documented in this book, not on anything you see in the Tcl scripts.

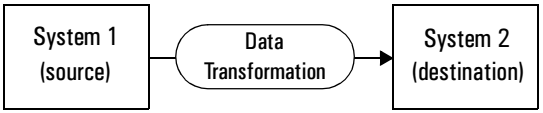
The Enterprise Link configuration tool allows the end user to define and modify a system's configuration information. The end user can define what data to transfer, how to transform the data values, where to transfer the data, and when to transfer the data. The configuration tool stores this configuration information in a configuration repository.

The Enterprise Link data server retrieves the configuration information from the configuration repository, then transfers the data between systems according to the configuration information:



Data Transfer

The Enterprise Link configuration tool allows the end user to define where a single unit of data comes from in one system (the *source* system), how the data is transformed, and where it is written to in a second system (the *destination* system).



Where a single unit of data comes from in one system (the source address), how it is transformed, and where it is written to in a second system (the destination address) is called a *mapping*:

{source address} — {transformation} → {destination address}

A mapping can define data transfer and transformation either for a single unit of data or for a set of dynamically configured sources and destinations. Both kinds of mappings are often organized into groups the configuration tool's user interface calls *methods*. This book refers to these *methods* as *configured methods*.

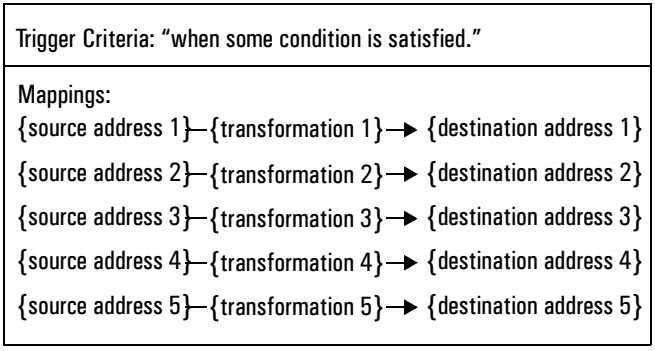
Configured Method
{source address 1} — {transformation 1} → {destination address 1}
{source address 2} — {transformation 2} → {destination address 2}
{source address 3} — {transformation 3} → {destination address 3}
{source address 4} — {transformation 4} → {destination address 4}
{source address 5} — {transformation 5} → {destination address 5}

Note

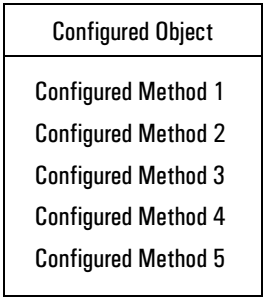
This book uses the term *method* in the usual object-oriented sense as applied to Tcl; namely, an action or function associated with an [incr Tcl] class that acts on an instance of the class.

A configured method contains not only mappings but also the configuration information that defines when to transfer the data. This configuration information is called a configured method's *trigger criteria*. Therefore, a configured method can be defined as one or more mappings that specify

what to transfer and where to transfer it, and one or more trigger criteria that specify when to transfer the data.



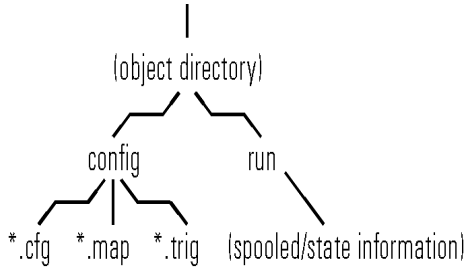
Just as mappings are grouped into configured methods, configured methods are organized into larger entities called *configured objects*.



The configuration tool creates these objects and the data server animates them. The configuration tool deals with only one configured object at any given time, while the data server animates configured objects one at a time.

Configured Objects

A configured object contains files and directories. The base directory is named after the configured object. This directory can be located anywhere in the file system, just as long as the configuration tool and data server have access to it.



- The `run` directory stores spooled data and state information that the data server needs at run time. The data server automatically creates the run directory the first time the directory is needed.
- The config directory contains the object's configuration files. The configuration tool automatically creates the `config` directory when the object is created. The data in each configuration file is stored in the form of a Tcl script. This makes it easy for the configuration tool and data server to load the data.
- The names of configuration files reflect what they contain. For example, a configuration file named errorlog.cfg contains configuration data that describes where the data server should log error messages. Configuration files containing mapping configuration data have the same name as their configured method but with a .map suffix. Configuration files containing trigger configuration data have the same name as their configured method but with a .trig suffix.

Some characters are permitted in configured method names but not in file names. The **utlFnameToStr** and **utlStrToFname** utilities can decode and encode configured method names that contain these special characters. For information about these utilities, see "utlFnameToStr" on page 8-38 and "utlStrToFname" on page 8-70.

The following example shows a generic layout of a configuration file. This configuration file assumes that the configuration data consists of two variables: <*what*>*_variable_one* and <*what*>*_variable_two*.

Example

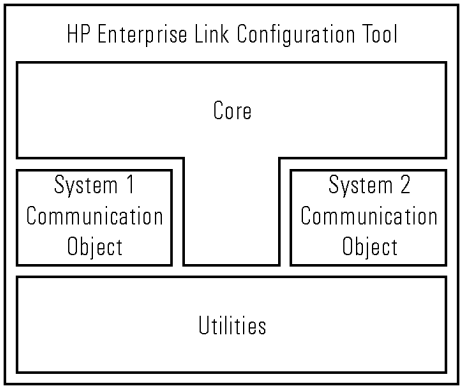
```
 1: # Enterprise Link <what> Configuration File
 2:
 3: if {![info exists _prefix]} {set _prefix "<what>_"}
 4:
 5: set ${_prefix}variables "${_prefix}config_rev \
        ${_prefix}variable_one ${_prefix}variable_two"
 6:
 7: set ${_prefix}config_rev 1
 8: set ${_prefix}variable_one <value one>
 9: set ${_prefix}variable_two <value two>
10:
11: # EOF
```

The lines in actual configuration files are not numbered. The line numbers were added to this configuration file to make it easier to explain each line.

- The first line is a comment that identifies what the file contains.
- The third line specifies a prefix to add to every variable name in the file. The existence of this prefix allows Tcl programs to avoid name collisions when loading multiple configuration files.
- The fifth line creates a variable containing a list of all the Tcl configuration data variables that sourcing this file will create. This variable is always named *<what>_variables*. Tcl programs that read configuration files may use or ignore this variable.
- The seventh line creates a variable containing a number that identifies the file format used for the configuration data. This variable is saved whenever Tcl programs write to the file and checked whenever Tcl programs read the file. The following file format changes cause this revision number to increment:
    - adding new configuration data
    - deleting existing configuration data
    - entering new possible values for variables that can store enumerated types
- The eighth and ninth lines create two configuration variables. Configuration files may contain one or more such entries.
- The eleventh line contains a comment to indicate the end of file. This line is useful for detecting truncated configuration files.

Configuration Tool Overview

The following diagram shows the high-level architecture for the Enterprise Link configuration tool.



The Enterprise Link configuration tool is composed of three important pieces:

1. A core component that implements all of the configuration tool's generic, system-independent functionality.
2. Two communication objects, each crafted specifically for a system such as Agilent Technolgies' RTAP product or SAP's R/3 product. These communication objects allow the configuration tool to display a customized user interface to the end user. The configuration tool must be provided with two such communication objects: one for each of the two systems that will be exchanging data.

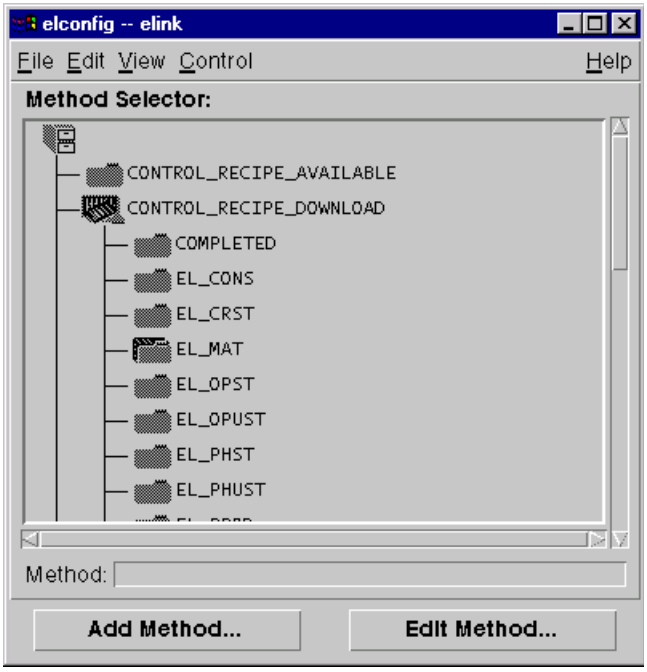
Note

The NULL Communication Object is a simple communication object that allows you to run the Enterprise Link configuration tool (`elconfig`) and the Enterprise Link data server (`elserver`) when you have only one complete communication object. Use the NULL communication object to run and test Enterprise Link while your own communication object is still in development. For more information on the name space and other important features of the NULL communication object, see the file `null_config.tcl` located in the directory `%ELROOT%\lib\NULL`.

3. A utility component that provides Tcl and Tk programmers with many useful routines for manipulating data and creating windows. You should use these utilities when you develop a communication object.

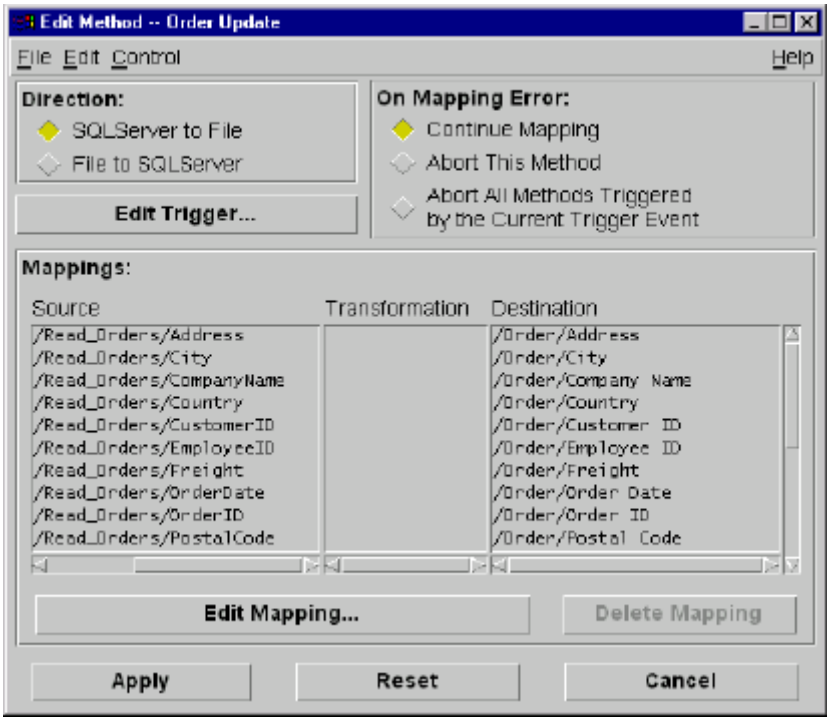
Configuration Tool User Interface

The configuration tool allows the end user to create new configured objects or modify existing ones. The name of the configured object currently open for editing is displayed in the title bar of the configuration tool's main window. The main window appears when the configuration tool is first started. The Open... menu item in the main window's File menu opens any configured object for editing.



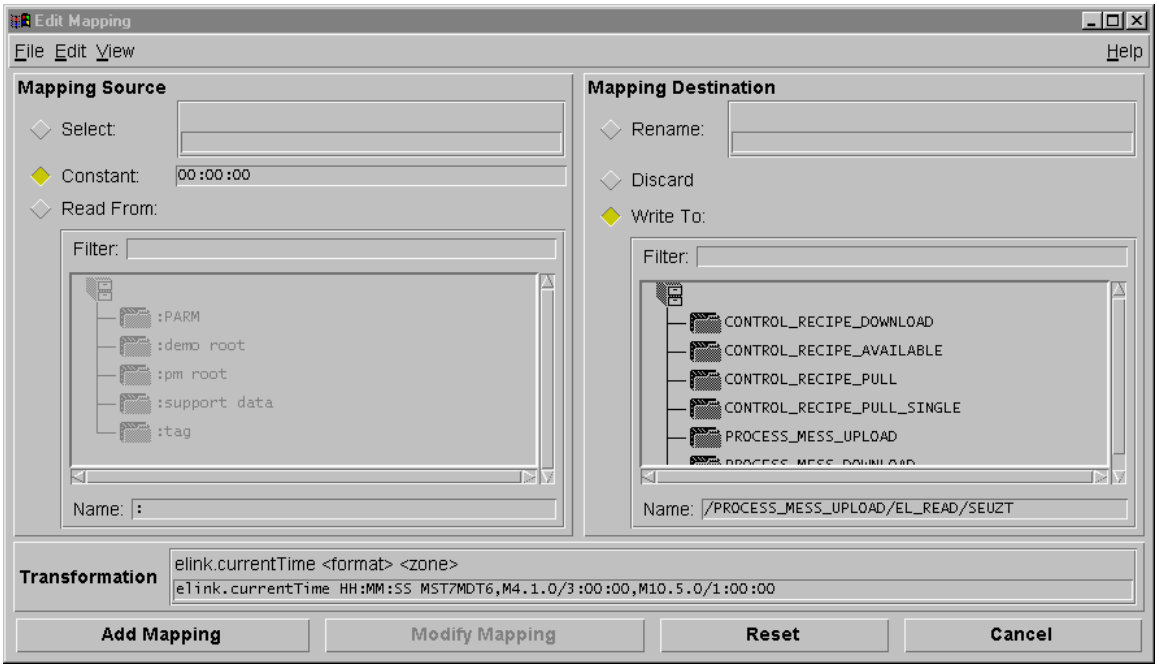
The configuration tool allows the end user to organize a configured object's configured methods into a hierarchy. These are displayed in the main window as nodes in the Method Selector diagram. This diagram appears blank if no configured methods are currently defined. To create a new configured method, click on the Add Method... button. This causes the Add Method window to appear. To edit an existing configured method, select the

appropriate node on the diagram then click on the Edit Method... button. This causes the Edit Method window to appear.



The Edit Method window displays mappings as a multicolumn list of source-destination pairs, along with any transformation expressions defined for the mappings. This list is empty if no mappings are currently defined, as is the case for newly created configured methods.

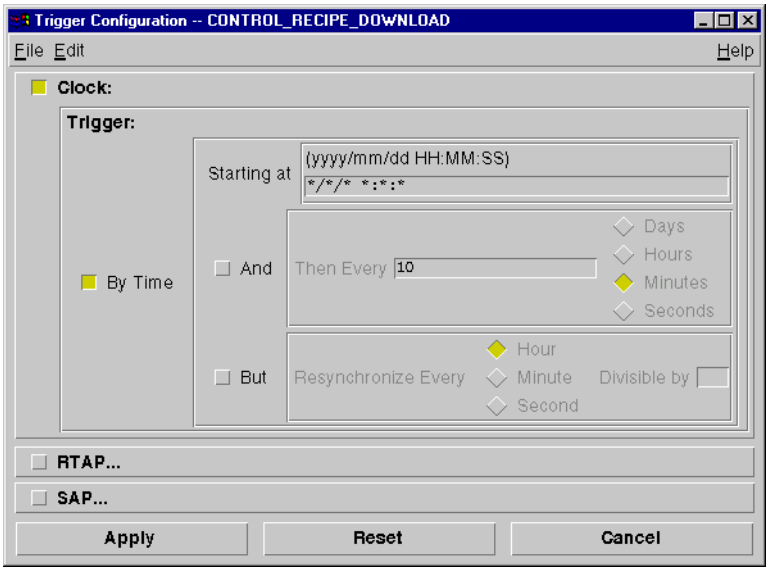
To define new mappings click on the Edit Mapping... button. To edit an existing mapping, select the existing mapping then click on the Edit Mapping... button. In either case, the Edit Mapping window appears.



The Edit Mapping window allows the end user to add new mappings to the selected configured method or to modify existing mappings. The left-hand side of this window displays all possible data sources, and the right-hand side of this window displays all possible data destinations.

The Transformation text-entry box allows you to specify a procedure or expression that will transform a value as it passes from one communication object to another. In the illustration above, the elink.currentTime procedure—one of the generic procedures included with Enterprise Link—is used to set the value for the current date and time in the Mountain Standard Time zone. For more information on generic procedures, see the *Enterprise Link User's Guide*.

To complete the creation of a configured method, specify trigger criteria by clicking on the Edit Trigger... button located near the top of the Edit Method window. This causes the Trigger Configuration window to appear.

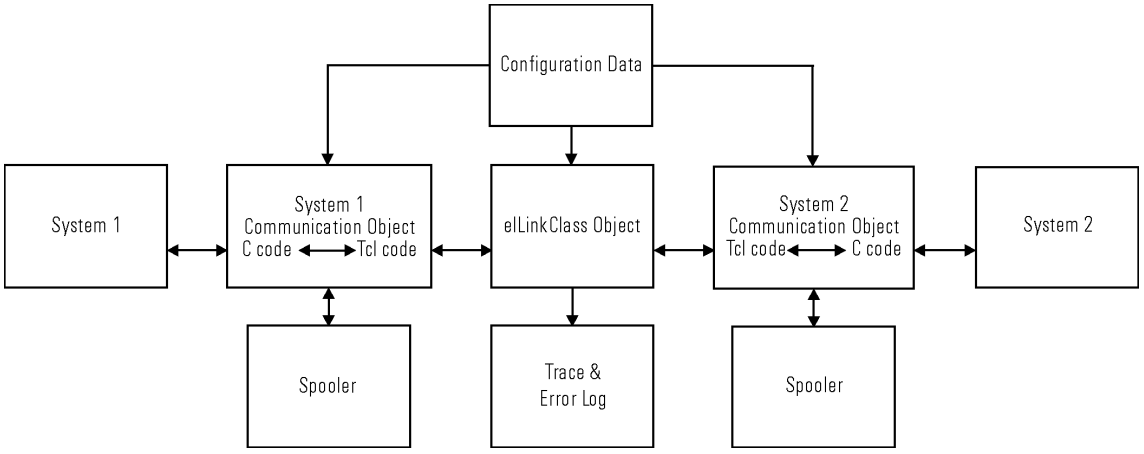


The Trigger Configuration window allows the end user to specify the circumstances that cause data to be transferred. The top portion of this window contains generic trigger criteria and the lower portion contains trigger criteria specific to each of the two systems that data is being transferred between.

For more detailed information on Enterprise Link's configuration tool, see the *Enterprise Link User's Guide*.

Data Server Overview

The data server's data flow looks generally like the following:



The **elLinkClass** object mediates all communication between the communication objects, and each communication object mediates all communication between its target system and the **elLinkClass** object. Each communication object is composed of both Tcl code and C code, with the C code responsible for direct communication with the target system's API. Each communication object may use spooler objects to store messages and other information that cannot be delivered immediately. The **elLinkClass** object also manages communication with the trace and error logs.

A communication class usually contains the methods provided by the base communication class **elCommClass**. For information about each method, see "elCommClass" on page 7-5.

You may add methods to your communication class to support or assist these standard methods. You can also omit methods from your class if the corresponding method in the base communication class **elCommClass** already does everything you need it to do.

The following describes the data server's control flow:

1. The elserver script initializes some environment variables and global variables, then invokes the elserver main function.
2. The main function creates the **elLinkClass** object and both communication objects. This invokes the **constructor** method in each object. The main function then passes control to the **elLinkClass** object.
3. The **elLinkClass** object invokes each communication object's **options** method to check for command-line option collisions. Collisions occur when a command-line option is used by two different communication objects, presumably to mean two different things.
4. If any collisions are detected, the **elLinkClass** object renames the collision-bound options and invokes **options** again in both communication objects, instructing the objects to use the renamed options.
5. The **elLinkClass** object then invokes **consumeOptions** in each communication object, instructing each object to remove any command-line options it recognizes from the command line and to remember them. Any remaining options are removed by the **elLinkClass** object. Any options that were not removed by **elLinkClass** or either communication object results in an error message.
6. If any command-line processing errors occurred, the **elLinkClass** object invokes **usage** in each of the communication objects and formats a human-readable command-line usage message. The usage message is printed for the end user after a description of the command-line error. The elserver script then exits.
7. If no command-line processing error occurred, the **elLinkClass** object invokes the **open** method in each communication object. The **open** method must first load the communication, spooling, or other configuration files it needs from the configuration directory. The **open** method must then use that information, and any command-line options that its **consumeOptions** method found, to initialize the communication object and prepare it for operation. If the communication object needs any "worker" objects like spoolers, the **open** method must create and initialize them.
8. The **elLinkClass** object loads each trigger file in the configuration directory, then invokes **setTrigger** in each communication object for each trigger. Each **setTrigger** method must examine the information loaded from the trigger file and decide if the information describes a

trigger that the communication object is responsible for detecting. If so, the communication object must create any event or message handlers that are required to detect the trigger, and if necessary, must communicate to its target system the need to generate those events or messages. Alternatively, the communication object could stop short of contacting the target system and instead allow the **run** method to communicate with the target system when the **run** method is invoked later. The communication object must also create any cross-reference information that is required to associate a configured method name with the occurrence of a trigger condition.

9. The **elLinkClass** object invokes the **run** method in each communication object. The **run** method completes any actions not already completed by the **setTrigger** method in order to set up triggers in the target systems. By default, the **setTrigger** method performs all required actions and the **run** method in the **elCommClass** responds accordingly.

10. After all the triggers have been set, elserver goes into an event-driven mode of operation.

11. When a communication object's event handler activates, and the object detects that a trigger condition has been satisfied, it gives the **elLinkClass::execute** method the names of all the configured methods that satisfied the trigger condition.

12. **elLinkClass::execute** determines which communication object is the source and which is the destination for each configured method. **elLinkClass::execute** then invokes **read** in each source communication object to acquire an array containing the source values for statically configured methods. If the source values are configured dynamically, **read** invokes the selection procedure associated with the configured mappings, notes the paths returned by each selection procedure mapping, and reads the values returned by those paths.

13. **elLinkClass::execute** then maps the source values to the destination paths—and transforms values and renames source paths if these options are selected—passing the result to the **write** method in each destination communication object. In the case of mappings that use selection and renaming procedures, **elLinkClass::execute** invokes the renaming procedure once for each of the paths returned by the selection procedure.

14. When all data for all the configured methods has been written to the destination communication objects, **elLinkClass::execute** invokes **commit** in each communication object that was written to with the **write**

method. For some target systems, such as RTAP, **commit** is a meaningless operation. For those communication classes, you do not need to write a **commit** method—the one in the **elCommClass** base class will respond correctly. With target systems such as an SQL database, the **commit** method makes permanent all the changes written to the target system with **write**.

15. If an error occurred during any **write** operation—or in any selection, renaming or transformation procedure—and if error handling is configured to abort all methods when that error occurs, **elLinkClass::execute** invokes **rollBack** in each communication object that was written to with the **write** method. For some target systems, **rollBack** is meaningless. For those communication classes, you do not need to write a **rollBack** method—the one in the **elCommClass** base class will respond correctly. For other communication classes, **rollBack** undoes all the changes written to the target system with **write** since the last **commit** or **rollBack**.
16. When the data server receives an exit command or a kill signal, it destroys all the communication objects. This invokes each object's **destructor** method. The data server then exits.

Note There can be more than one **write** method in a configured method.

Configuration Tool/Data Server Interface

To write the communication classes in Tcl, you must decide what information the communication objects in the configuration tool will exchange with those in the data server. Currently there are only two kinds of information you need to decide on:

- trigger configuration
- access configuration

Trigger configuration is the information an end user enters into the configuration tool's Trigger Configuration window, and it describes when a configured method should be executed.

Access configuration is the information an end user enters into the configuration tool's Access Configuration window, and it provides the information that the communication object may need to contact the target system—for example, an IP address, the name of the target system, a user ID and password, a serial port name, speed or parity configuration, and so on.

Both kinds of information are communicated from the configuration tool to the data server through configuration files in the config directory. The files are designed to be loaded directly into the data server Tcl environment with the Tcl source command. The most important information in each configuration file is a number of set x y commands that have the result of setting local variables in any Tcl procedure that loads the file.

For the trigger and access configuration files, you have several decisions:

- Whether your target system can trigger the execution of configured methods and if so
  - which variables will store the trigger information, and
  - what values should those variables have and what do the values mean?
- Whether your communication object needs any user-specified information to contact your target system and if so
  - which variables will store the user-specified information, and
  - what values should those variables have and what do the values mean?

Tcl/C API Interface

As described earlier, developing communication objects involves three main activities:

1. extending the Tcl interpreter,
2. writing a communication class for the data server, and
3. writing a communication class for the configuration tool.

Software development can generally proceed on all three activities simultaneously once the interfaces between these components are defined. You must define the interface between the C-coded Tcl interpreter extensions and the two Tcl-coded communication objects. The interface is simply the set of Tcl commands and their arguments that the C extension will make available to Tcl programmers.

If the interface to your target system is very simple—for example, an ASCII protocol on a tty port or a socket—you may not need to write any C-code. You may be able to use the standard Tcl file manipulation commands, or one of the more popular sockets extension packages like TclX or DP to send messages to and receive messages from your target system.

If you need to write a custom Tcl extension, then your interface design is constrained by whatever communication mechanisms are available to communicate with your target system. You'll have a hard time doing any more than your target system's API lets you do. However, a common mistake in defining the interface is to create a one-to-one correspondence between Tcl commands and a complex suite of API functions or communication primitives. When you create the Tcl interface, you have an opportunity to simplify the mechanisms for communicating with your target system, which makes writing the Tcl portion of your communication object much easier. For example, if the mechanics of sending a message to and receiving a reply from your target system require you to invoke 5 or 10 C functions for one transaction, you should encapsulate that complexity in one command:

```
set result [sendMessage destination message]
```

or perhaps two commands:

```
sendMessage destination message
set result [getReply destination]
```

However, the desire to simplify the interface must be balanced against the cost of simplification. Most people write Tcl code faster than an equivalent amount of C code. Therefore, you don't want to spend too much effort

encapsulating complexity in C, since it would be more cost effective to do complex encapsulation in Tcl. Therefore, if simplifying the interface results in a large body of C code, you may want to expose a slightly more complex interface in Tcl commands and build the final simplifying layer on top of those commands with Tcl code.

Another factor to weigh in the design of your interface is whether you want the interface to your target system to be message-oriented or database-oriented:

- In message-oriented systems, the communication object communicates with its system by sending it messages and receiving messages from it. Paths in the object's logical name space identify fields in those messages.
- In database-oriented systems, the communication object communicates with its system as if it was a database. Paths in the object's logical name space identify values in the database.

If you can find a way to interpret your target system as a database-oriented system, do so. Writing interfaces for database-oriented systems is generally easier than for message-oriented systems. The orientation of your target system to a large extent determines your interface. A database-oriented system generally looks like the following example with db_read returning a list of values, one for each address:

```
db_write whichDatabase {{address value} {address value} ...}
db_read whichDatabase {address1 address2 ...}
```

A message-oriented system generally looks like the following example:

```
sendMessage destination message
set result [getReply destination]
```

Message-oriented systems tend to be more complex than database-oriented systems because messages and replies generally consist of a great many values sent and received simultaneously. These values must all be supplied at once, must be formed into a message using the right formats, and must be in the right order. Database-oriented systems are much more of a random-access mechanism, where you can send each datum as it comes without coordinating it into a larger message.

Yet another factor to weigh is the requirement for the data server to respond promptly to external stimuli. Enterprise Link exchanges high-level information between systems, so sub-millisecond response times are generally not required. However, someone may be waiting for information to be exchanged, so subsecond response is often desirable. To respond

promptly to external stimuli, the data server should not stop and wait a long time for a communication object that is waiting for a response from its system.

Ideally, every operation that may take a long time to complete should be carried out asynchronously, and some kind of event handler should be invoked when the operation is complete. This way, the data server can continue processing other events while the communication object completes the operation. This tends to be easier to do in a message-oriented communication object because each message can be considered an independent transaction.

In practice, the asynchronous approach is difficult to implement for a database-oriented communication object and becomes more difficult if the target system's API prohibits it. If you get subsecond response from your target system, you may choose to live with the delays and any potential throughput degradation. If you have multisecond response times, you have little choice but to use an asynchronous design. If the API prohibits a straightforward asynchronous design, you must emulate such behavior by starting a slave process that talks to the target system, potentially being delayed for a long time, while the data server continues processing other events.

All of these issues should be considered when designing the interface between your C code and your Tcl code. Once the interface—the Tcl command set—is defined, work can proceed in parallel on the C and the Tcl portions of the communication object. For testing purposes, you may wish to write some Tcl stubs to emulate the commands in the interface until the C code is complete.

This book explains only the initial steps of using the build script to create your own Tcl interpreter. For details on writing C code to extend your Tcl interpreter, see *Tcl and the Tk Toolkit* by John K. Ousterhout (Reading, Massachusetts: Addison-Wesley Publishing Company, 1994).

The *Enterprise Link Developer's Guide* also describes writing the Tcl portion of the data server and configuration tool communication classes (see chapter 2, "Building a Tcl Interpreter," chapter 3, "Developing the Data Server Communication Object," and chapter 4, "Developing the Configuration Tool Communication Object").

Tcl Communication Class Concepts

This section provides Tcl communication class concepts that you must be familiar with. Some of these concepts will affect the design of your communication objects.

[incr Tcl]

The data server and portions of the configuration tool are written using [incr Tcl], an object-oriented extension to the Tcl programming language. The extension lets you define classes and methods, use inheritance and instantiate objects with a syntax reminiscent of C++. If you are not already familiar with [incr Tcl], you should review the incrTcl(3) man page. If you are using Enterprise Link on Windows NT, you can access this man page information through the file %ELROOT%\tcl\7.6\src\itcl-1.5.tar.

Error Logging and Tracing

The **elLinkClass** object provides a **log** method that communication objects must use to write error, warning, and tracing information to the error and tracing logs. The **elLinkClass** object reads the logging configuration, decides where and how big the logs should be, and so forth. In particular, **elLinkClass** keeps track of which kinds of messages should be logged and which should not. Methods in communication classes should invoke the **log** method any time they have anything to report or to trace. The **log** method will either send the message to the appropriate log or will discard the message depending on how the end user configures it.

Briefly, there are six message types the **log** method understands:

error	Something has gone wrong and data probably has been or will be lost.
warning	Something has gone wrong but the data server was able to recover successfully. Data is not lost.
verbose	Status information for debugging communication objects.

in	Prints a copy of input received from the target system.
out	Prints a copy of output going to the target system.
to	Prints a copy of path/value pairs sent to a communication object through its **write** method. This type is not something you'll invoke in your code. It is only invoked by **elLinkClass::execute**. See "elLinkClass::log" on page 7-26 for more details.

The **log** *in* **/log** *out* style of input and output tracing is an important diagnostic function built into communication objects, while *to* tracing happens automatically in the data server. Together, these three types of tracing follow the three major stages of data progress through Enterprise Link:

1. *in* traces data as it enters the source communication object from the source system.
2. *to* traces data as it moves from the source communication object to the destination communication object.
3. *out* traces data as it leaves the destination communication object.

Because many vendors are usually involved in an Entprise Link deployment, these three stages of tracing help you determine which vendor is responsible when data either doesn't make it through, or gets corrupted inside Enterprise Link.

Discarding Values

End users can configure communication objects to discard information just before it is sent to the corresponding target system or just after it is received from the target system. Discarding information lets you debug communication objects and methods without fear that you will send bad data to the target system. The configuration information that controls discarding data is read automatically by the **elCommClass::open** method and is available to your methods in the **$discardOutput** and **$discardInput** variables.

When the **$discardOutput** variable is 1, communication objects must not send data from **write** methods to the target system. Ideally, nothing at all should be sent to the target system, but this is not always possible. There may be some "handshake" or "heartbeat" messages that must be exchanged with the target system, especially if input is still being accepted. Output should always be traced with an elLink log out... invocation before being discarded.

When the **$discardInput** variable is 1, communication objects must discard any data received from the target system. Ideally, all data is discarded, but again, this is not always possible. There may be "handshake" or "heartbeat" messages that must be acted upon to keep the communication link with the target system open, especially if output is still being sent to the target system. Input should always be traced with an elLink log in... invocation before being discarded (see "Error Logging and Tracing" on page 1-21).

When the **$discardInput** and **$discardOutput** variables are both 1, communication objects must not even attempt to open a connection to their target system. This setting must be interpreted to mean the target system is probably not even present and the data server is being operated strictly in a debugging mode. The data server must function even when it can't communicate with the target system for which it was instructed to discard both input and output.

Spooling Data

If your communication object and your target system will be separated by a less-than-reliable WAN, or if your target system itself is less than reliable, you must build spooling capability into your communication object. Spooling allows you to buffer values written to your object while the network or target system is unavailable. The data server provides three classes to help you spool information:

- The **elSpoolerClass** is the base class for the other two spooler classes. It contains an **open** method that knows how to load a standard spooler configuration file and a few small utility functions that let you query the configuration from outside a spooler object (see "elSpoolerClass" on page 7-34).
- The **elFIFOSpoolerClass** implements a first-in first-out (FIFO) spooler. You can append character-string messages to the spooler and retrieve them later (see "elFIFOSpoolerClass" on page 7-21).
- The **elRASpoolerClass** implements a random-access spooler. You must specify an "ID tag" for each message you spool, and you can retrieve and delete messages by specifying the tag (see "elRASpoolerClass" on page 7-30).

Logical Name Space Interpretation

A logical name space is the set of all the character-string names for addresses in the system from which data values can be received or to which data values can be sent. Enterprise Link has little built-in knowledge of how information is organized in any system. The product simply assumes that some subset of the values in the system can be addressed using character string names and that for some systems the naming scheme is hierarchical.

Names of values in the logical name space are called *paths* because most end users are familiar with the UNIX or Windows NT file systems, which represent one kind of hierarchical logical name space. For example, if you build a UNIX or Windows NT file system communication object, values would be the contents of files and paths would be the pathnames of those files. A path has two representations:

- a human-readable representation
- a Tcl-list representation

The human-readable representation is the conventional representation for a path. The Tcl-list representation has a Tcl list element for the name in each level in the path. For example, if the human-readable representation of a path were /usr/mail/andrew, then the corresponding representation as a Tcl list would be {usr mail andrew}.

If your logical name space is not hierarchical, then your path has only one level. Every value will have a name and all those names will be displayed as one, possibly very long, list in the configuration tool's main window. As a rule, if you have several hundred names in your name space, you should impose some sensible hierarchy on the names to improve usability. In the configuration tool, it is much easier to manipulate several hundred or several thousand names in a hierarchical path than it is to manipulate a single long list of these names.

In addition to deciding how to convert the human-readable paths into Tcl lists and vice-versa, you must also determine the meaning of paths in the logical name space. In message-oriented systems, paths in the name space usually look like the following and identify fields in messages:

```
/messageName/fieldName
```

In database-oriented systems, paths in the name space usually identify values in the database. A relational database path may look like the following:

```
/schemaName/tableName/fieldName
```

You must determine what format for paths makes sense for your target system. Then you must implement that format in your communication class and document it so your end users can create configured methods.

If your target system has tables or messages that are directly addressable in the name space—as opposed to being hidden deep in the implementation of your communication class—you must answer the following questions:

1. How do I know when a message is complete and can be sent?
2. How do I know when a row in a table is complete and I can start putting data in the next row?
3. If I need to map data from different rows in the same table into different destinations, how can I do that?
4. If I need to map data from the same kind of message into different locations depending on what is in the message, how can I do that?

Questions 1 and 2 are answered by the **write** method in the communication class. When you execute a configured method, **write** is invoked once for each mapping to a communication object. In all communication classes with

directly addressable messages and tables in the logical name space, each **write** invocation specifies one complete message or table row. In general, if your **write** method was not passed enough information to populate a complete message or table row, you should log an error and inform the end user that the configured method is incomplete.

The exception to this rule is when a directly addressable table is contained in a message. If it only makes sense to send one such message of any given type for any given trigger, each **write** method can add a row to the table and the **commit** method can send the message on its way. If you see a need to send multiple messages, each containing a different table, then you must implicitly identify the message that belongs to a given row of values in a configured method.

The best way to identify the message is to find or synthesize a field in the message outside of the table that is guaranteed to be different for every different message that could be triggered by the same trigger. If you require your end users to specify a value for this field in every configured method that writes to the table, then you can use that field to determine to which message the row values in the **write** invocation belong. You can create new messages/tables for every different value and send them all when **commit** is invoked.

Questions 3 and 4—about mapping values from different rows into different destinations—are answered by adding to your Trigger panel in the configuration tool. If the configured method you execute depends on the values in a row or in a message, the Trigger panel must contain fields to let you specify those fields and values. For example, you could define a Trigger panel that looks like the following:

> Trigger the configured method when:
> the value of path XXX in the target system changes AND
> Table TTT, field YYY has the value ZZZ

If you leave YYY and ZZZ empty, then the configured method is executed every time the value of path XXX in the target system changes, assuming, of course, that you can persuade the target system to notify the communication object when the value of that path changes. If you specify YYY and ZZZ differently in a number of different configured methods, then you are saying to execute each method only on the rows in the table TTT whose field/value data match the criteria laid out in the Trigger panel.

2

Building a Tcl Interpreter

Building a Tcl Interpreter

To extend the Tcl interpreter, you must first build your own version of the Tcl interpreter supplied with Enterprise Link, then write C code to extend your version of the interpreter. This chapter covers only the first part, providing instructions to build your own Tcl interpreter and describing the behavior of the build script. Writing the C code necessary to extend the Tcl interpreter is not covered in this book. For that information, see *Tcl and the Tk Toolkit* by John K. Ousterhout.

To build your own Tcl interpreter, you must have the Enterprise Link developer's component (ELINK-TCLDEVEL) installed. If you do not already have the component installed on your system, see the *Enterprise Link Installation Guide*.

The ELINK-TCLDEVEL component provides the following items:

- A build script that builds a Tcl interpreter using the files provided by the ELINK-TCLDEVEL component.
- Copies of the Tcl public distribution source and patch files that were used to build the Tcl interpreter shipped with Enterprise Link.
- Source files (elMain.c on HP-UX and elsh.c on Windows NT) that can be customized to incorporate new Tcl libraries and commands.

On HP-UX systems, the ELINK-TCLDEVEL component provides the following additional items:

- A setup script that creates a writable copy of the build directory.
- A copy of the GNU public distribution program patch that allows you to apply patches to the Tcl public distributions.
- The include file crStandards.h that is needed by the source file elsh.c.
- The archived library libelsh.a that contains Enterprise Link functions, including compiled versions of the source files elMain.c and elsh.c.
- The archived library libtcllic.a that contains needed licensing functions.

On Windows NT systems, the ELINK-TCLDEVEL component provides the following additional items:

- Windows NT/Intel binaries for the GNU programs tar, patch, and sed, as well as cygwin.dll, a DLL required by these programs. If required, source code for these programs can be obtained from ftp://ftp.cygnus.com/pub/gnu-win32/latest. The program tar is used to unarchive the tar files, and sed is used to edit some of the Tcl makefiles.
- elRFC.obj, an object file generated with Microsoft Visual C++ 4.2 containing Enterprise Link Tcl bindings for SAP R/3 RFCs.

Building a Tcl Interpreter on HP-UX Systems

1. Install the Enterprise Link developer's component ELINK-TCLDEVEL (see the HP-UX version of the *Enterprise Link Installation Guide* for instructions).
2. Prepare for the build by creating a writable copy of the build directory:

```
cd
${TCLROOT}/src/Setup
```

3. Change to your newly created build directory and execute the build script:

```
cd myBuildDir
./Build
```

The first run of the build script may take up to half an hour since it must unpack and compile all the Tcl source files. After the build script successfully completes, you will have a basic Tcl interpreter named elsh.

Note

If you don't have the products RTAP and SAP installed on your system, the new elsh Tcl interpreter will not support the missing products.

4. If the build fails because you have an older version of the GNU program patch on your system, clean up the build directory and rerun the build script using the -patch option:

```
./Build -clean
./Build -patch
```

This option causes the build script to build patch version 2.2, and then use patch to apply patches. Older versions of patch have a defect that the Tcl public distribution patch files encounter. This defect causes problems when applying publicly distributed patches to these source files.

5. Customize elsh.c, elMain.c, and build to incorporate your new Tcl commands and libraries.
6. Rerun the build script to build a customized Enterprise Link Tcl interpreter.
7. After you build a working Enterprise Link Tcl interpreter, rerun the build script using the -clean option to clean up your build directory:

```
./Build -clean
```

Building a Tcl Interpreter on Windows NT™ Systems

Note

If you intend to use the SAP Communication Object, you must already have SAP R/3 RFC libraries installed on your system in order for the newly built elsh to support this product.

1. Install the Enterprise Link Tcl interpreter component ELINK-TCL and developer's component ELINK-TCLDEVEL (see the Windows NT version of the *Enterprise Link Installation Guide* for instructions).
2. Create your own development directory and copy all files and subdirectories from %TCLROOT%\src to your development directory.
3. Add your new development directory and its subdirectory, gnu-win32, to the environment variable PATH. Set environment variables through the System Properties window's Environment tab. You can access the System Properties window by opening the control panel then by double-clicking on the System icon.
4. Open a command prompt window and change to your new development directory.
5. Run the command .\Build to create a basic Tcl interpreter called elsh.exe.

   If this command fails, run the command .\Build -clean followed by .\Build.

Note

If another user installed Visual C++ on your system, you should check the Include environment variable to make sure it contains the directory path C:\msdev\include, and that the Lib environment variable contains the directory path C:\msdev\lib, assuming that Visual C++ is installed in C:\msdev.

6. Customize the elsh.c and build files in order to incorporate your new Tcl commands and libraries. The build file is a Tcl script.
7. Rerun the build script in order to build the customized Enterprise Link Tcl interpreter.
8. Once you have built a working Enterprise Link Tcl interpreter, rerun the build script with the -clean option to clean up your build directory.

The Build Script

The build script included with the Enterprise Link Tcl developer's component does most of the work required to build a Tcl interpreter. This section describes what gets compiled and linked when you execute the build script, and how you can use the build script's compile and link options.

Include Files and Symbolic Constants

For compiling, the build script tells the compiler where to find the include files needed to build a Tcl interpreter. On both HP-UX and Windows NT systems, these files are in the directories *myBuildDir/*include, *myBuildDir/*tk, and *myBuildDir/*tcl.

On HP-UX systems, the build script tells the compiler where to find the X11 include files. For HP-UX 10.20, these files are under the directory /usr/include/X11R5.

If you have SAP installed on your system, the build script also defines the C preprocessor symbolic constant INIT_SAP to include SAP's RFC functionality.

On HP-UX systems, the build script defines the C preprocessor symbolic constant INIT_LIC to include Enterprise Link licensing functionality.

You can tell the compiler where to find include files by using the **-I** command-line option or, on HP-UX systems, by using the HP-UX environment variable CCOPTS. Similarly, you can define C preprocessor symbolic constants using the **-D** command-line option. On Windows NT systems, for example, the **-D** command-line option could look like the following:

```
cl -Ox -W3 -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo -D_X86_=1
-DWINVER=0x0400 -DWIN32 -D_WIN32 -D_MT -D_DLL -Ic:\msdev\include
-I..\win -I..\generic -D__WIN32__ -DUSE_TCLALLOC=0 -Dtry=__try
-Dexcept=__except -c -ImyBuildDir\include -ImyBuildDir\tk4.2\generic
-ImyBuildDir\tcl7.6\generic -Fo.\ myBuildDir\elsh.c
```

While on HP-UX systems, the **-D** command-line option could look like this:

```
c89 -O -c \
-DelTCL_HOME=/opt/tcl -DelTCL_VERSION=7.6 \
-DelINIT_SAP -DelINIT_LIC \
-ImyBuildDir/include -ImyBuildDir/tk -ImyBuildDir/tcl \
-I/usr/include/X11R5 applic.c
```

If you are compiling on HP-UX from the command line, you can shorten this command by setting CCOPTS appropriately. If you are running either a Bourne or a Korn shell, the syntax is

```
    CCOPTS='-DelTCL_HOME=/opt/tcl -DelTCL_VERSION=7.6 \
    -DelINIT_SAP -DelINIT_LIC \
    -ImyBuildDir/include -ImyBuildDir/tcl \
    -I/usr/include/X11R5'; export CCOPTS

c89 -O -c applic.c
```

If you are running a C shell, the syntax is

```
    setenv CCOPTS '-DelTCL_HOME=/opt/tcl -DelTCLVERSION=7.6 \
    -DelINIT_SAP -DelINIT_LIC \
    -ImyBuildDir/include -ImyBuildDir/tcl \
    -I/usr/include/X11R5'

c89 -O -c applic.c
```

Tcl Libraries

After compiling, the build script builds the Tcl libraries in the current directory, *myBuildDir*.

On HP-UX systems, the build script also builds the Tcl libraries in the Tcl source directory, *myBuildDir*/tcl.

On Windows NT systems, the build script builds the Tcl libraries in the following additional directories:

- the Tcl source directory: myBuildDir\tcl7.6\win
- the Tk source directory: myBuildDir\tk4.2\win
- the TclX source directory: myBuildDir\tclx7.6.0\win
- the [incr] Tcl source directory: myBuildDir\itcl-1.5\win

After building the libraries, the build script tells the compiler to search for and link to the Tcl libraries.

If you did not run the build script provided by the ELINK-TCLDEVEL component, but you want to build a custom Tcl interpreter, you must tell the compiler to search for and link to the Tcl libraries that were used to build the elsh interpreter shipped with Enterprise Link. These Tcl libraries are located in the directory /opt/tcl/7.6/lib on HP-UX systems, and in the directory %TCLROOT%\lib on Windows NT systems. These directories are included in the ELINK-TCLDEVEL component.

On HP-UX systems, the build script tells the compiler to search for and link to X11 libraries. For HP-UX 10.20, the X11 libraries are usually found in the directory /usr/lib/X11R5.

SAP RFC Libraries on HP-UX Systems

If you have SAP installed on your system, the build script tells the compiler to search for and link to SAP RFC libraries to include SAP's RFC functionality. For HP-UX 10.20, the SAP RFC libraries are usually found in the directory /opt/sap/3.0c/rfcsdk/lib.

- The build script tells the compiler to search for then link to the product licensing library in the directory /opt/rtap/A.07.00/shlib or in the directory /opt/rtap/E.01.20/shlib.
- The build script tells the compiler to link to the following libraries in the order shown:
    - The archived library libelsh.a that contains Enterprise Link functions.
    - The archived library libtcllic.a that contains Enterprise Link licensing functions.
    - Optionally the archived library librfc.a that contains any needed SAP RFC functions.
    - The archived library libtcl.a that contains any needed Tcl interpreter functions.
    - The archived library libX11.a that contains any needed X11 functions.
    - The math library libm.a.

You can tell the compiler to search for and link to libraries using the **-L** command line option. You can tell the compiler to link to libraries using the **-l** command line option. For example,

```
c89 -O -Wl,-E \
-o myBuildDir/elsh myBuildDir/applic.o \
-LmyBuildDir -LmyBuildDir/tcl
-L/opt/tcl/7.6/lib -L/usr/lib/X11R5 \
-L/opt/sap/3.0c -L/opt/sap/3.0c/rfcsdk/lib \
-lelsh -ltcllic -lrfc \
-ltcl -lX11 -lm
```

SAP RFC Libraries on Windows NT™ Systems

If you have SAP R/3 RFC libraries installed on your system, the build script tells the compiler to search for and link to the following libraries and object files in order to include SAP's RFC functionality:

- **elRFC.obj**: an object consisting of Tcl bindings for the SAP R/3 RFC interface.
- **libtcl.dll**: a DLL for Tcl version 7.6, this file contains any needed Tcl interpreter functions.
- **librfc32.dll**: a DLL for SAP R/3 RFC functions.
- **System DLLs**: msvcrt.dll, oldnames.dll, kernel32.dll, advapi32.dll, user32.dll, gdi32.dll, comdlg32.dll, and winspool.dll.

For example,

```
link /NODEFAULTLIB /INCREMENTAL:NO /PDB:NONE /RELEASE /NOLOGO
-align:0x1000 -subsystem:console,4.0 -entry:mainCRTStartup
myBuildDir\elsh.obj myBuildDir\elRFC.obj
myBuildDir\tcl7.6\win\libtcl.lib c:\sap\3.0c\rfcsdk\lib\librfc32.lib
msvcrt.lib oldnames.lib kernel32.lib advapi32.lib user32.lib gdi32.lib
comdlg32.lib winspool.lib -out:myBuildDir\elsh.exe
```

3

Developing the Data Server Communication Object

Developing the Data Server Communication Object

This chapter describes how to develop a communication object for the Agilent Technologies Enterprise Link data server.

To develop a Tcl communication object for the data server, you write methods for a communication class. Agilent Technologies recommends that you add the following points into your development of the data server communication object:

- Ensure that you do not enable the data server to create or display a window.
- Ensure that the **elserverPath** contains the full directory pathname of the elserver process. You can use this variable to determine which bin directory elserver will reside in.
- In order for the data server to support trigger tracing, ensure that the communication object calls the following Tcl procedure whenever the communication object triggers a method:

```
elLink log trigger $this {[
   set traceMsg "<my type of trigger>"
   <collect/compose additional communication object specific
   information here>
   append tracemsg "<additional information>"
   set traceMsg
]}
```

Data Server Methods

The following methods are typically used in the development of communication objects for data servers. All of the methods except the **supports** method are optional. If your communication class does not support the functionality in the method, you do not need to develop the method since the **elCommClass** base class provides default methods that respond correctly for unsupported functionality. To take advantage of the default methods, make your communication class inherit from the **elCommClass** base class.

Example

```
itcl_class myCommClass {
   inherit elCommClass
...
}
```

Method Name	Description	Page
supports	Indicates whether or not a feature is supported.	3-9
selectionProcedures	Retrieves, deletes, executes and creates selection procedures.	3-10
constructor	Initializes instance variables.	3-14
destructor	Shuts down communication with the target system.	3-15
list2path	Converts a Tcl list to a path string.	3-16
path2list	Converts a path string to a list.	3-17
options	Queries and sets command-line option keywords.	3-18
consumeOptions	Processes command-line arguments.	3-19
usage	Returns command-line usage information.	3-20
open	Loads configuration information and prepares the object for use.	3-21
setTrigger	Sets a trigger for an indicated method.	3-23
run	Invoked by the data server after all triggers have been passed to **setTrigger**.	3-23
getChildren	Returns the child node names as a Tcl list.	3-25
read	Reads values from the target system.	3-26
write	Writes values to the target system.	3-29
commit	Makes all **write** invocations since the last **commit** or **rollBack** permanent.	3-31
rollBack	Undoes all **write** invocations since the last **commit** or **rollBack**.	3-32
getSpoolPaths	To support the Diagnostic GUI display of spool files in your own communication object.	3-33

Enterprise Link Communication Object Methods

All of the methods of an Enterprise Link communication object that change the current working directory during their execution must be sure to restore the current working directory to its original value before returning. Failure to do this will result in undesirable behavior of the elserver process. For example, it will no longer be possible to stop the elserver process from the interactive configuration tool elconfig.

The current working directory must be restored whenever any communication object method returns successfully, unsuccessfully, and whenever a Tcl error occurs during method execution.

Note that the current working directory can be obtained by calling the Tcl procedure `pwd` and can be changed by calling the Tcl procedure `cd`.

The Tcl method shown below demonstrates how to use the Tcl *catch* procedure to ensure that the current working directory is always restored to its original value:

```
method write args {

     # --- note the original working directory ---
     set orig_pwd [pwd]

     # --- do the write inside a "catch" command ---
     if {[catch {

               :
          cd <the desired directory>
               :
          <do write-related stuff here>

     # --- restore current working directory ---
     } rv]} {
          catch {cd $orig_pwd}
          error $rv
     } else {
          cd $orig_pwd
     }

     return $rv
}
```

Note

It's usually faster to write Tcl code than to write C code because in Tcl you avoid C's compile-link-execute cycle. However, the resultant Tcl code generally runs one hundred to one thousand times slower than a corresponding C function. If you expect your communication objects to do a lot of low-level data manipulation, you may choose to write their most expensive parts in C rather than in Tcl. If you're not sure how expensive the Tcl code will be, you can always write it first in Tcl and then later translate into C some or all of whatever turns out to be the most expensive.

To develop a new communication object for the data server, do the following:

Step 1: Develop an [incr Tcl] method to indicate which features are supported.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::supports
```

For a description of how to develop this method, see "Developing the supports Method" on page 3-9.

Step 2: Optionally develop an [incr Tcl] method to retrieve, execute, create and delete user-configured selection procedures for the communication object.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::selectionProcedures
```

For a description of how to develop this method, see "Developing the selectionProcedures Method" on page 3-10.

Step 3: Optionally develop an [incr Tcl] method to carry out non-trivial initialization of instance variables.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::constructor
```

For a description of how to develop this method, see "Developing the constructor Method" on page 3-14.

Step 4: Optionally develop an [incr Tcl] method to shut down communication with the target system.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::destructor
```

For a description of how to develop this method, see "Developing the destructor Method" on page 3-15.

Step 5: Optionally develop an [incr Tcl] method to convert a Tcl list to a path string.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::list2path
```

For a description of how to develop this method, see "Developing the list2path Method" on page 3-16.

Step 6: Optionally develop an [incr Tcl] method to convert a path string to a Tcl list.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::path2list
```

For a description of how to develop this method, see "Developing the path2list Method" on page 3-17.

Step 7: Optionally develop three [incr Tcl] methods to query and set command-line option keywords, search a list of command-line arguments, and return command-line usage information.

For example, if the target system type is MYSYS, the methods should have the following names:

```
elServerMYSYSClass::options
elServerMYSYSClass::consumeOptions
elServerMYSYSClass::usage
```

For a description of how to develop these methods, see "Developing the options Method" on page 3-18, "Developing the consumeOptions Method" on page 3-19, and "Developing the usage Method" on page 3-20.

Step 8: Optionally develop an [incr Tcl] method to load the configuration file and set the instance variables.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::open
```

For a description of how to develop this method, see "Developing the open Method" on page 3-21.

Step 9: Optionally develop an [incr Tcl] method or methods to set a trigger for the indicated method.

For example, if the target system type is MYSYS, the method or method pair should have the following names:

```
elServerMYSYSClass::setTrigger
elServerMYSYSClass::run
```

For a description of how to develop the **setTrigger** method or the **setTrigger** and **run** method pair, see "Developing the setTrigger and run Methods" on page 3-23.

Step 10: Optionally develop an [incr Tcl] method to return child node names as a Tcl list.

For example, if the target system type is MYSYS, the method should have the following name:

```
elServerMYSYSClass::getChildren
```

For a description of how to develop this method, see "Developing the getChildren Method" on page 3-25.

Step 11: Optionally develop four [incr Tcl] methods to read values from the target system, write values to the target system, and commit or roll back write invocations in target systems that support a commit concept.

For example, if the target system type is MYSYS, the methods should have the following names:

```
elServerMYSYSClass::read
elServerMYSYSClass::write
elServerMYSYSClass::commit
elServerMYSYSClass::rollBack
```

For a description of how to develop these methods, see "Developing the read Method" on page 3-26, "Developing the write Method" on page 3-29, "Developing the commit Method" on page 3-31, and "Developing the rollBack Method" on page 3-32.

Step 12: Integrate the new communication object into the data server.

1. Copy the elserver file located in the bin directory to a different directory.
2. Edit the elserver file and replace all references to one of the communication objects (usually RTAP) with the name of your object and all references to the class of that object (usually RTAP_class) with the class of your object.
3. Adjust your PATH environment variable so that it finds your elserver file before the default elserver file.

Developing the supports Method

You must develop the **supports** [incr Tcl] method. This method indicates whether or not dynamic mapping and time-based triggers are supported. The data server invokes this method at startup to determine what functionality the communication object provides and what data server facilities are needed. When the data server invokes **supports**, it passes this method one parameter: a keyword identifying a unit of functionality that the data server wants to know about. The expected return value is either 0 or 1. The **supports** method must recognize the following keywords:

selection procedures Indicates whether or not the communication object supports dynamic mapping. A return value of 1 allows the communication object's **read** method to invoke selection procedures.

time triggers Indicates whether or not the communication object supports time-based triggers. If the communication object does not support time-based triggers, the data server handles all time-based triggering.

Unrecognized keywords should cause the **supports** method to return 0, which means the unrecognized unit of functionality is not supported.

The following example of a **supports** method shows that the communication object does support dynamic mapping, but does not support time-based triggers.

Example

```
method supports args {

   set l [llength $args]

   # if `keyword' is omitted, return a list of
   # recognized keywords
   if {$l == 0} {
      return [list "selection procedures" "time triggers"]

   # else if `keyword' is present
   } elseif {$l == 1} {
      set k [lindex $args 0]
      switch $k {
               "selection procedures" {return 1}
                      "time triggers" {return 0}
                             default {if {$debug} \
                                      {error "Unrecognized keyword: $k"}}
      }

   # else more than one argument has been supplied, error
   } else {
      utlArgEnd
   }

   return 0
}
```

Developing the selectionProcedures Method

If you want your communication object to support dynamic mapping, you must develop the **selectionProcedures** [incr Tcl] method. This method gets, deletes, executes or creates selection procedures. The **selectionProcedures** method has four modes of operation.

1. Gets

In its first mode, **selectionProcedures** takes no arguments and returns a list of selection procedures supported by the source communication object, which includes selection procedures originating in both the communication object and the data server.

The returned list contains three elements: the procedure name, procedure arguments, and the application programming interface (API) version of the selection procedure. For the E.02.20 version of Enterprise Link, the API version should always be 1.

Example

```
if {[llength $args] == 0} {
   set rc {}
   foreach proc_name [info procs {xxx_select_*}] {

         regsub "^xxx_select_(.*$)" $proc_name {\1} p_name

         # --- fetch procedure's API version number ---
      if {[info exists select_api_versions($p_name)]} {
         set p_version $select_api_versions($p_name)
      } else {
         set p_version 0  ;# version is unknown
      }

         # --- properly convert default selection procedure arguments ---
         set p_args {}
         foreach arg_name [info args $proc_name] {
            if {[info default $proc_name $arg_name arg_value]} {
               lappend proc_args [list $arg_name $arg_value]
            } else {
               lappend proc_args $arg_name
            }
         }

      lappend rc [list $p_name $p_args $p_version]
   }

   return $rc
}
```

2. Deletes

In its second mode of operation, the **selectionProcedures** method takes any number of selection procedure names as arguments and returns a Tcl list of the names of deleted procedures.

Example

```
set rc {}
foreach arg $args {

   set arg_length [llength $arg]

   if {$arg_length == 1} {
      set p_name [lindex $arg 0]

      rename xxx_select_$p_name {}

      unset select_api_versions($p_name)
      if {[array size select_api_versions] == 0} {
         unset select_api_versions
      }

      lappend rc $p_name
   }
}
```

3. Executes

In its third mode of operation, the **selectionProcedures** method takes any number of two-item arguments. The first list item is a selection procedure name and the second is a list of that selection procedure's arguments.

This third mode of **selectionProcedures** returns a Tcl list of zero or more three-item lists. The first item is the selection procedure executed, along with the selection procedure's arguments. The second item is the error code generated from the execution. The third item is a list of selected source paths, with each path in the form of a Tcl list. If the procedure execution is successful, the item in the list representing the error code will be empty and the last item will contain selected source paths. If the procedure execution was unsuccessful, the error code will appear in the list and the following item representing source paths will be empty.

Example

```
set rc {}
foreach arg $args {

   set arg_length [llength $arg]

   # --- just an execute-procedure request? ---
   if {$arg_length == 2} {
      set p_name [lindex $arg 0]
      set p_args [lindex $arg 1]

      # --- set defined arguments ---
      set el_args [list $this]

      set sources {}

      if {[info exists select_api_versions($p_name)]} {

         # --- if no errors occur, "err_code" item is empty ---
         set err_code {}

         set api_version $select_api_versions($p_name)

         # --- invoke selection procedure appropriately ---
         switch -- $api_version {
            "0" -
            "1" {if {[catch {set sources [eval xxx_select_$p_name \
            $el_args $p_args]}]} {
                  global errorInfo errorCode
                  set err_code [list $errorInfo $errorCode]
                 }
         }

         default {set err_code [utlNls "API version mismatch error: \
         selection-procedure `%0' requires unsupported API version #%1" \
         $p_name api_version]
         }

      } else {
         set err_code [utlNls "Undefined selection-procedure \
         (procedure=%0)" $p_name]
      }

      lappend rc [list $arg $err_code $sources]
   }
}
```

4. Creates

In its fourth mode, the **selectionProcedures** method takes any number of arguments in a four-item list. The first item is a selection procedure name. The second item is a list of that selection procedure's arguments. The third item is the body text defining the selection procedure, and the fourth item is the application programming interface (API) version of the selection procedure. In the E.02.20 version of Enterprise Link, the API version is 1.

This fourth mode of the **selectionProcedures** method returns a Tcl list of zero or more procedure names.

Example

```
set rc {}
foreach arg $args {

   set arg_length [llength $arg]

   if {$arg_length == 4} {
      set p_name [lindex $arg 0]
      set p_args [lindex $arg 1]
      set p_body [lindex $arg 2]
      set p_vers [lindex $arg 3]

      # --is this an unsupported version of the selection-procedure API? --
      if {$p_vers > 1} {
         error [utlNls "Unsupported version of the selection-procedure \
         API (procedure=%0 version=%1)" $p_name $p_vers]
      }

      proc xxx_select_$p_name $p_args $p_body
      set select_api_versions($p_name) $p_vers
      lappend rc $p_name
   }
   return $rc
}
```

Developing the constructor Method

You can optionally develop the **constructor** [incr Tcl] method. Most communication objects will not have a **constructor** method, which is necessary only if you need to carry out non-trivial initialization of instance variables. For example, if you have an instance variable that is an array and you need to populate the array with some constants initially, you would do that in the **constructor**. If you need to initialize scalar instance variables, you can do that in the declaration of the variable.

For example, if you have a message-oriented communication class and logical name-space paths all have the form {messageName fieldName}, then you may need a fieldNames instance variable that is an array. This array converts message names into the list of fields defined for the message. The **constructor** could initialize that array.

Example

```
itcl_class myCommClass {
   inherit elCommClass

   protected numericVar 23       # initial value is 23
   protected listVar {a b c}     # initial value is {a b c}
   protected fieldNames          # an array variable

   # Initialize arrayVar
   method constructor {
      set fieldNames(messageName1) {fieldName1 ... N}
      set fieldNames(messageName2) {fieldName2 ... N}
      ...
   }
   ...
}
```

Developing the destructor Method

You can optionally develop the **destructor** [incr Tcl] method. The **destructor** method should release all the resources used by your object. This should include any other objects your object created, any communication channels your object opened, any temporary files for your object, and so on.

Developing the list2path Method

You can optionally develop the **list2path** [incr Tcl] method. This method converts a Tcl list into a path string specification. A similar method must be developed for the configuration tool (see "Developing the list2path Method" on page 4-12).

The syntax of the path string specification should be appropriate for the type of target system. For example, if the type of target system is Agilent Technolgies' RTAP product, path specifications would use the same syntax as RTAP's symbolic database addressing.

The data server invokes the **list2path** method to convert Tcl lists to ordinary path string specifications. The data server displays the paths to the end user as ordinary path strings but performs all computations on paths when they are encoded as Tcl lists.

When the data server invokes **list2path**, it passes this method one parameter: the Tcl list to be converted to a path string specification. It expects the **list2path** method to return the resulting path string specification.

The following example of a **list2path** method assumes that the path string syntax is similar to that used by the UNIX file system—that is, item names separated by solidus characters (/).

Example

```
method list2path p {
   return utlList2Path $p
}
```

Developing the path2list Method

You can optionally develop the **path2list** [incr Tcl] method. This method converts a path specification into a Tcl list. A similar method must be developed for the configuration tool (see "Developing the path2list Method" on page 4-13).

The data server invokes **path2list** to convert ordinary path string specifications to paths expressed as Tcl lists. This method must implement exactly the opposite conversions that are implemented in the **list2path** method.

When the data server invokes **path2list**, it passes this method one parameter: the path string to be converted to a Tcl list. It expects the **path2list** method to return the resulting Tcl list.

The following example of a **path2list** method assumes that the path string syntax is similar to that used by the UNIX file system; that is, item names separated by solidus characters (/).

Example

```
method path2list p {
   return utlPath2List $p
}
```

Developing the options Method

You can optionally develop the **options** [incr Tcl] method. This method gets or sets the command-line option keywords supported by the communication object. A similar method must be developed for the configuration tool (see "Developing the options Method" on page 4-14).

When starting the data server, you can type in command-line options on your operating system's command line; for example, **-file** <*name*>. Command-line option keywords, **-file** in this example, distinguish one command-line option from another.

The data server invokes **options** up to two times during startup. The first invocation is to get the list of default option keywords supported by the communication object. If any of those keywords are also used by another communication object or by the data server's core, this method is invoked a second time to rename those keywords. When the data server invokes **options** the first time, it does not pass in any parameters. It expects the **options** method to return a Tcl list of default command-line option keywords. When the data server invokes **options** the second time, it passes in one parameter: a Tcl list with the new (possibly changed) values for the communication object's command-line option keywords. The data server expects the **options** method to return a Tcl list of these new command-line option keywords. The number of items that the data server includes in this Tcl list will always equal the number of items that were returned by the data server's first invocation of the **options** method.

The following example of an **options** method assumes that the communication object supports two command-line options: **-option1** *name* (an option with one argument) and **-option2** (an option with no arguments).

Example

```
itcl_class myCommClass {
   inherit elCommClass
   protected option1Keyword -option1
   protected option2Keyword -option2
   ...
   method options {args} {
      if {[llength $args] > 0} {
         set optkeys [lindex $args 0]
         if {[llength $optkeys] == 2} {
            lassign $optkeys option1Keyword option2Keyword
         }
      }
      # Return the current command-line option keywords.
      return [list $option1Keyword $option2Keyword]
   }
```

Developing the consumeOptions Method

You can optionally develop the **consumeOptions** [incr Tcl] method. This method parses and consumes the command-line options contained in a specified global variable. A similar method must be developed for the configuration tool (see "Developing the consumeOptions Method" on page 4-15).

The data server invokes this method during startup to provide the communication object with an opportunity to parse its command-line options in the data server command line.

At startup, the data server invokes this method, passing it one parameter: the name of the global variable containing the program's command-line options as a Tcl list. The **consumeOptions** method should examine this list and remove any options that apply to the communication object. All unrecognized options should be left untouched since the data server passes the resulting list to the other communication object, then consumes all command-line options supported by the core. If any command-line options remain after this, the data server reports a usage error and exits.

To parse for and remove recognized command-line options, use the **utlGetArg** utility (see "utlGetArg, utlPeekArg, utlArgEnd" on page 8-43). You may use the **utlPeekArg** utility to examine, but not remove, a command-line option that the data server's core supports and will eventually remove.

In the following example, the **consumeOptions** method parses and consumes the options supported in the example for the **options** method.

Example

```
method consumeOptions {varName} {
   upvar $varName args

   # Fetch command-line options.
   set opt1 [utlGetArg $option1Keyword {default value}]
   set opt2 [utlGetArg $option2Keyword]
}
```

Developing the usage Method

You can optionally develop the **usage** [incr Tcl] method. This method returns the text for a command-line usage message. The data server invokes this method whenever command-line usage errors are detected. The data server uses the information returned by this method to compose and print a human-readable usage message. A similar method must be developed for the configuration tool (see "Developing the usage Method" on page 4-16).

The returned value is a Tcl list with one entry per command-line option. Each entry is itself a list with two elements: the option string and an English explanation of the option. If the English explanation will not fit on one line, you may indent the text by inserting new-line characters and space characters. The data server converts all new-line characters followed by space characters in the text into a single space character.

In the following example, the **usage** method returns the text in the example for the **options** method.

Example

```
method usage {} {
   return "
      {{\[$option1Keyword <name>]} {An option with a
         parameter. Feel free to run this description over
         multiple lines.}}
      {{\[$option2Keyword]}                {A flag option.}}"
}
```

Developing the open Method

You can optionally develop the **open** [incr Tcl] method. This method prepares a communication object for operation. It should load any required configuration file from the configuration directory. A similar method must be developed for the configuration tool (see "Developing the open Method" on page 4-17).

At some point you are likely to override this method in your communication class. When this happens, you must invoke the base class **open** method, which will load the data-discarding configuration for the object:

```
elCommClass::open
```

The method in the base class initializes the base-class instance variables:

```
protected discardInput
protected discardOutput
```

These variables are set to 0 by default, and to 1 when the communication object has been configured to discard input from or output to the target system (see "Discarding Values" on page 1-23).

If you are building a configuration tool access window for the target system, this method must load the access configuration file that the window produces. This method must then do any initialization or communication channel creation associated with the information in the access configuration file. If it is possible to open a connection to your target system using only the defaults in the configuration tool's access window, do not assume that the access configuration file exists in your **open** method.

If your class supports data spooling, the **open** method should initialize the spoolers as well.

Example

```
..

protected destAddr 23;    # default connection parameters
protected destMode auto

protected connectionID;    # for connection to your target system

method open {} {
   elCommClass::open   ;# invoke the base class open

   # Load the "access" config file
   set configFileName [elLink configDir]/${this}_access.cfg
   if {[file exists $configFileName]} {
      if {[catch {source $configFileName} errMsg]} {

         elLink log error $this \
            {Cannot load $configFileName, must exit: $errMsg}
         exit 1
      } else {
         set destAddr [set ${this}_access_destination_address]
         set destMode [set ${this}_access_destination_mode]
      }
   }

   # If we're discarding both input and output, we're done -- do
   # NOT connect to the target system.
   if {$discardInput && $discardOutput} {
      return
   }

   # Open a connection to the target system
   if {[catch {set connectionID \
         [openConnection $destAddr $destMode]} errMsg]} {
      elLink log error $this \
         {Cannot open connection to target system: $errMsg}
      exit 1
   }

   # Figure out the send spooler's config file name. It's
   # something like XXX2${this}_spooling.cfg.
   set files *2${this}_spooling.cfg
   catch {set files [glob $files]}
   if {[llength $files] > 1} {
      set msg "Using [lindex $files 0] as the `*2${this}'"
      append msg " spool config file: $files"
      elLink log warning $this {$msg}
      set files [lindex $files 0]
   }

   # Start the send spooler.
   if {[catch {elFIFOSpoolerClass $this.sendSpooler \
         $files ${this}SendSpool} errMsg]} {
      elLink log error $this \
         {Could not start send spooler: $errMsg}
      catch {$this.sendSpooler delete}
   }
}
```

Developing the setTrigger and run Methods

You can optionally develop the **setTrigger** [incr Tcl] and **run** [incr Tcl] methods. These methods can be set up in either of two ways:

- the **setTrigger** method runs every time it receives trigger information, or
- the **setTrigger** method collects trigger information until the **run** method is invoked by the **elLinkClass::execute** method.

In the first case, the **setTrigger** method performs all the activities described in this section and the default **run** method does nothing. In the second case, the activities are split between the **setTrigger** and **run** methods.

In both cases, the **setTrigger** method associates a configured method name with a trigger condition. If necessary, the **setTrigger** method—or, in the second scenario, the **setTrigger** and **run** method pair—tells the target system to notify the communication object that a trigger condition occurred and establishes a Tcl event handler to receive the notification.

In both cases, the **setTrigger** method takes a single argument: the configured method name with which the trigger condition is associated. If **$discardInput** is set to 1, ignore invocations to this method (see "Discarding Values" on page 1-23).

The invoker's local variables contain the description of the trigger condition. Your **setTrigger** method should use the `upvar` command to extract the values of these variables. The names and meanings of these variables are part of the interface you defined between the configuration tool and the data server. See "Configuration Tool/Data Server Interface" on page 1-17.

In the first case, the setTrigger method completes the trigger implementation in your communication object by requesting the invocation of a Tcl handler function whenever a trigger is detected. This function or method does not have a reserved name and can be given any name you want. The handler function needs to examine the triggers set in the object and invoke the **elLinkClass::execute** method to execute all the methods whose trigger conditions were met.

For example, if you have a message-oriented communication class and receiving a message is the only way to trigger a method execution, then your **setTrigger** method and handler function could look like the following example.

Example

```
...
protected triggers            # trigger(msgName) = {method1 ... N}
protected lastMsg {}          # The last message the app sent us
protected lastMsgName {}

method setTrigger {methodName} {

   # Set no triggers if input is being discarded.
   if {$discardInput} {
      return
   }

   upvar ${this}_trigger_msg_enable         msgEnable \
         ${this}_trigger_msg_name           msgName

   #If
   if {$msgEnable} {
      lappend triggers($msgName) $methodName
   }
}

# The message handler method that is invoked every time a message
# from the target system is received. "message" is a list of
# values.
method msgHandler {msgName message} {

   # Tell the user a message was received
   elLink log in $this {$message}

   set lastMsg $message
   set lastMsgName $msgName
   elLink execute $triggers($msgName)
}
```

If you want to use the **setTrigger** and **run** method pair, you can set up the **run** method to complete the trigger implementation in your communication object by requesting the invocation of a Tcl handler function whenever a trigger is detected. To do this, you could append the following lines to the example given above:

Example

```
method run {} {

      set desiredMessageNames [array names triggers]

      solicit_messages_from_application $desiredMessageNames

   }
```

Developing the getChildren Method

If your communication object supports dynamic mapping, you must develop the **getChildren** [incr Tcl] method. This method returns the names of children under a specified node that are actually available for the target system's name space.

The data server invokes this method whenever it needs to get the names of child nodes. When the data server invokes **getChildren**, it passes this method the full path name of the node (as a Tcl list) whose child names are wanted. This method should return a Tcl list of child names that appear directly under this node. The returned child names should not include any path information.

The optional parameter **-filter** <*pattern*> specifies a pattern that filters child names. Only child names that match <*pattern*> should be returned. If this parameter is omitted, all child names should be returned.

By definition, the root node of the name space tree is the empty string.

Example

```
method getChildren args {

   # fetch optional parameter
   set fil [utlGetArg "-filter" "*"]

   # fetch required parameter
   set n [utlGetArg]
   utlArgEnd

   if {$debug} {puts stderr "DEBUG: $this getChildren $args"}

   return < compose a Tcl list of the nodes under `$n' matching
          `$fil' here >

   }
```

Developing the read Method

You can optionally develop the **read** [incr Tcl] method. This method reads values from the target system and stores them in the source array. This method is passed a list of method names for which to acquire data and the name of the source array in which to store the data. When the triggering criteria are complex, the structure of the source array is complex (see "elCommClass::read" on page 7-12).

The **elLinkClass::execute** method invokes your **read** method when it needs data values from your object in order to execute the list of configured methods. Your **read** method must use **elLinkClass::methodInfo** to retrieve the list of name space paths that specify which data values are needed. Your method must also determine which, if any, selection procedures need to be invoked, then invoke those selections procedures and note the paths they return. Your **read** method must then retrieve any needed data values from the target system.

In database-oriented communication objects, the **read** method must query the database for the indicated source paths. In message-oriented communication objects, the **read** method must find source values for all the indicated source paths in the most recent message received from the target system.

In message-oriented communication classes, the **read** method is the only method that can tell if any received data was not mapped anywhere. When such data is detected, the **read** method must issue a warning (see "Discarding Values" on page 1-23).

The following examples show how to implement a simple read for a message-oriented communication class and for a database-oriented communication class.

Example 1

```
# The message-oriented read method

...
protected lastMsg {}            ;# The last message the app sent us
protected lastMsgName {}

method read {methodNames srcArray} {

   upvar $srcArray src
   upvar ${srcArray}_select sel

   # Do nothing if input is being discarded.
   if {$discardInput} {
      return
   }

   # Populate the source array
   foreach field $fieldNames($lastMsgName) value $lastMsg {
      set src([list $lastMsgName $field]) $value
   }

   # Populate the selected-paths array
   foreach method $methodNames {
      foreach pair [elLink methodInfo select $method] {
         lassign $pair destinationInfo sourceInfo
         set sel($sourceInfo) [execute_selection_procedure $sourceInfo]
      }
   }

   # Check for unused values.
   foreach field $fieldNames($lastMsgName) {
      set unused([list $lastMsgName $field]) {}
   }
   foreach method $methodNames {
      foreach pair [elLink methodInfo variable $method] {
         lassign $pair destination source
         catch {unset unused($source)}
      }
      foreach sourceInfo [array names sel] {
         foreach path $sel($sourceInfo) {
            catch {unset unused($path)}
         }
      }
      foreach path [elLink methodInfo discard $method] {
         catch {unset unused($path)}
      }
   }
}
```

Example 2

```
# The database-oriented read method

method read {methodNames srcArray} {
   upvar $srcArray src
   upvar ${srcArray}_select sel

   # Do nothing if input is being discarded.
   if {$discardInput} {
      return
   }

   # Read the requested values from the database
   foreach method $methodNames {
      foreach pair [elLink methodInfo variable $method] {
         lassign $pair destination source
         set src($source) [read_from_app_database [$this list2path
         $source]]
      }
      foreach pair [elLink methodInfo select $method] {
         lassign $pair destinationInfo sourceInfo
         set paths [execute_selection_procedure $sourceInfo]
         set sel($sourceInfo) $paths
         foreach path $paths {
            set src($path) [read_from_app_database [$this list2path $path]]
         }
      }
   }

   # Log the values read from the database
   if {[array size source]} {
      elLink log in $this {[
         set trace_msg {Read from the $this database:}
         foreach p [array names src] {
            catch {append trace_msg "\n    $p$src($p)"}
         }
         set trace_msg]}
   }

}
```

Developing the write Method

You can optionally develop the **write** [incr Tcl] method. This method sends data to the target system. In communication classes with no concept of a **commit** method, the values are written to the target system immediately. In classes with a **commit** method, the values are only written to the target system after the **commit** method is invoked or the values are undone when the **rollBack** method is invoked.

The **write** method's argument is the name of a variable in the invoking procedure that contains a list of *{destinationPath value}* pairs.

This method is invoked once for every configured method that executes. When values are written to a table, your **write** method should interpret all writes to the same table as being to the same row of that table. The next **write** invocation can start another row. In message-oriented communication objects, when values are written to a message, your **write** method should interpret all writes to the same message as being to the same instance of that message. The next **write** invocation can start another message. For messages that contain tables, the next **write** invocation generally starts a new row in the table and the message is not sent to the target system until **commit** is invoked.

If an error is encountered within the **write** method, you can discover how to deal with the errors using the following:

```
set how [elLink methodInfo error [elLink curMethod]]
```

A how value of

- continue means log the error and write the other values in the list to the target system.
- abandonMethod means log the error and return, without writing anything to the target system.
- abandonAllMethods means do *not* log the error, but raise a Tcl error condition with an informative error message. Your invoker is always **elLinkClass::execute**. It will log the error, abandon work on all other methods, and invoke **rollBack** for every object that was written to.

For more information on errors, see "Error Logging and Tracing" on page 1-21 and "Discarding Values" on page 1-23.

The following example shows a write method for a message-oriented communication class where paths in the name space looks like {messageName fieldName}.

Example

```
# A worker method for error handling
method errorMsg {msg} {
  if {[elLink methodInfo error [elLink curMethod]] == continue} {
    elLink log error $this {$msg}
  } else {
    error $msg
  }
}

# The message-oriented "write" method
method write {varName} {
  upvar $varName values ;# {{path1 value1} ... {pathN valueN}}
                        ;# where path = {messageName fieldName}

  # Build a random-access array of values:
  # raValues(path) = value
  #
  # and figure out how many different messages are being sent
  foreach pair $values {
    lassign $pair path value
    lassign $path messageName fieldName
    set raValues($path) $value
    set messages($messageName) {}
  }

  # Build each message:
  #  messages(messageName) = {value1 ... N}
  #
  # Note--fieldNames() is an instance variable in the communication object
  # describing what fields are in each message and in what order:
  #   fieldNames(messageName) = {field1 ... fieldN}
  foreach messageName [array names messages] {
    foreach field $fieldNames($messageName) {

      # Complain if any fields are missing
      set path [list $messageName $field]
      if {[catch {set value $raValues($path)}]} {
        errorMsg "You must specify a value for [list2path $path]"
        set value {}
      }
      lappend messages($messageName) $value
    }
  }

}

  # Log the values about to be sent to the system
  foreach messageName [array names messages] {
    elLink log out $this {$messages($messageName)}
  }

  # If we're discarding output, we're done
  if {$discardOutput} {
    return
  }

  # Send each message to the target system
  foreach messageName [array names messages] {
    sendToApplication $messageName $messages($messageName)
  }
}
```

Developing the commit Method

You can optionally develop the **commit** [incr Tcl] method. This method makes permanent in the target system the effects of all **write** invocations since the last **commit** or **rollBack**. If your target system does not support this concept, you do not need to define this method.

This method should work hard to avoid raising a Tcl error. If **elLinkClass::execute** has already invoked **commit** in several communication objects and then your communication object signals an error, very little error recovery can occur. **elLinkClass::execute** has no choice but to log an error and continue committing, because objects that have been committed cannot be rolled back.

Example

```
method commit {} {
   commitChangesInApplication
}
```

Developing the rollBack Method

You can optionally develop the **rollBack** [incr Tcl] method. In communication objects that support it, the **rollBack** method undoes the effects of all the **write** invocations since the last **commit** or **rollBack** invocation. If your target system does not support the concept, you do not need to define this method.

Example

```
method rollBack {} {
   rollBackChangesInApplication
}
```

Developing the *Object*_getSpoolPaths Procedure

To support the diagnostic GUI display of spool files in your own communication object, you must develop this new procedure. The ***Object*_getSpoolPaths** procedure determines the name and data-flow direction of each spool file associated with the communication object for a particular configuration.

Each `fname` is the absolute path of a spool file, and each `direction` is either `incoming` or `outgoing`.

Example

```
proc <obj>_getSpoolPaths {configDir runDir} {
    return [list <fname1> <direction1> <fname2> <direction2> ...]
}
```

For an example of the *Object*_getSpoolPaths procedure, see the file `SAP_spool.tcl`, located in the directory `%ELROOT%\lib\SAP`.

4

Developing the Configuration Tool Communication Object

Developing the Configuration Tool Communication Object

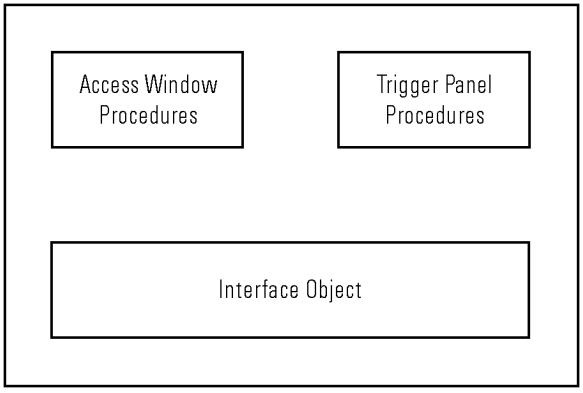
This chapter describes how to develop a communication object for the Enterprise Link configuration tool. In addition, this chapter describes how to write the methods for the configuration tool's interface class.

The configuration tool can be customized to support new target systems by developing new communication objects. Each communication object provides the following functionality to the configuration tool:

- Displays a view into the target system's name space, which allows the end user to more easily specify where data should be read from and where it should be written to.
- Creates data mappings to and from the target system.
- Optionally specifies system-specific triggers.
- Optionally specifies system-specific access parameters such as host computer names, login names, and passwords.

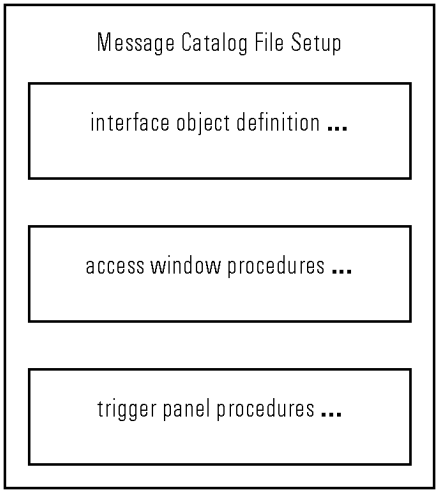
The communication object is composed of three important components:

- an interface object
- a collection of trigger panel procedures
- a collection of access window procedures



In the current version of Enterprise Link, trigger panel and access window routines are defined as procedures. This is necessary because Tk and [incr Tcl] do not currently work well together. In future versions of Enterprise Link, these procedures will eventually become methods.

The components of a communication object are typically defined in a single source file. A suggested source file layout is shown in the following figure.



A template source file is provided in the following directory:

```
<environment variable ELROOT>/lib/template/template_config.tcl
```

Note

It is usually faster to write Tcl code than to write C code because Tcl avoids C's compile-link-execute cycle. However, the resulting Tcl code generally runs 100 to 1000 times slower than a corresponding C function. If you expect your communication objects to do a lot of low-level data manipulation, you may choose to write their most expensive parts in C rather than in Tcl. If you're not sure how expensive the Tcl code will be, you can always write it first in Tcl and then later translate into C some or all of whatever turns out to run too slowly.

To develop a new Enterprise Link configuration tool communication object, follow these steps.

Step 1: Create a new configuration tool interface object (see "Developing the Interface Object" on page 4-6).

You must always create a configuration tool interface object. This object provides the configuration tool with a basic interface to the target system.

Step 2: Optionally create an access window for the target system (see chapter 5, "Developing an Access Window").

Creating the access window is optional. If you need to be able to define access data that is unique to the target system, you'll have to develop an access window.

Step 3: Optionally create a trigger panel for the target system (see chapter 6, "Developing a Trigger Panel").

Creating the trigger panel is optional. If you need to support trigger criteria that are unique to the target system, you'll have to develop a trigger panel.

Step 4: Optionally create a message catalog file for the communication object (see "Creating a Message Catalog File" on page 4-26).

Creating a message catalog file is optional. Message catalog files allow you to easily change displayed text to support localization.

Step 5: Integrate the new communication object into the configuration tool.

1. Create a directory in the path $ELROOT/lib (`%ELROOT%\lib` on Windows NT systems) and name the directory for your communication object. For example, $ELROOT/lib/SAP.
2. Copy the set of files for your communication object to the new directory.
3. Create new configured objects with your communication object as either the source or destination.

Developing the Interface Object

The configuration tool uses interface objects to interface with the target system. An interface object contains methods that can do the following.

- Indicate the functionality that the communication object supports.
- Handle system-specific configuration tool command-line parameters.
- Open a view into the target system's name space.

Viewing the target system's name space is an important feature of the configuration tool. The configuration tool displays a system's name space as a tree diagram. These system-specific tree diagrams are presented to the end user in the configuration tool's Edit Mapping window.

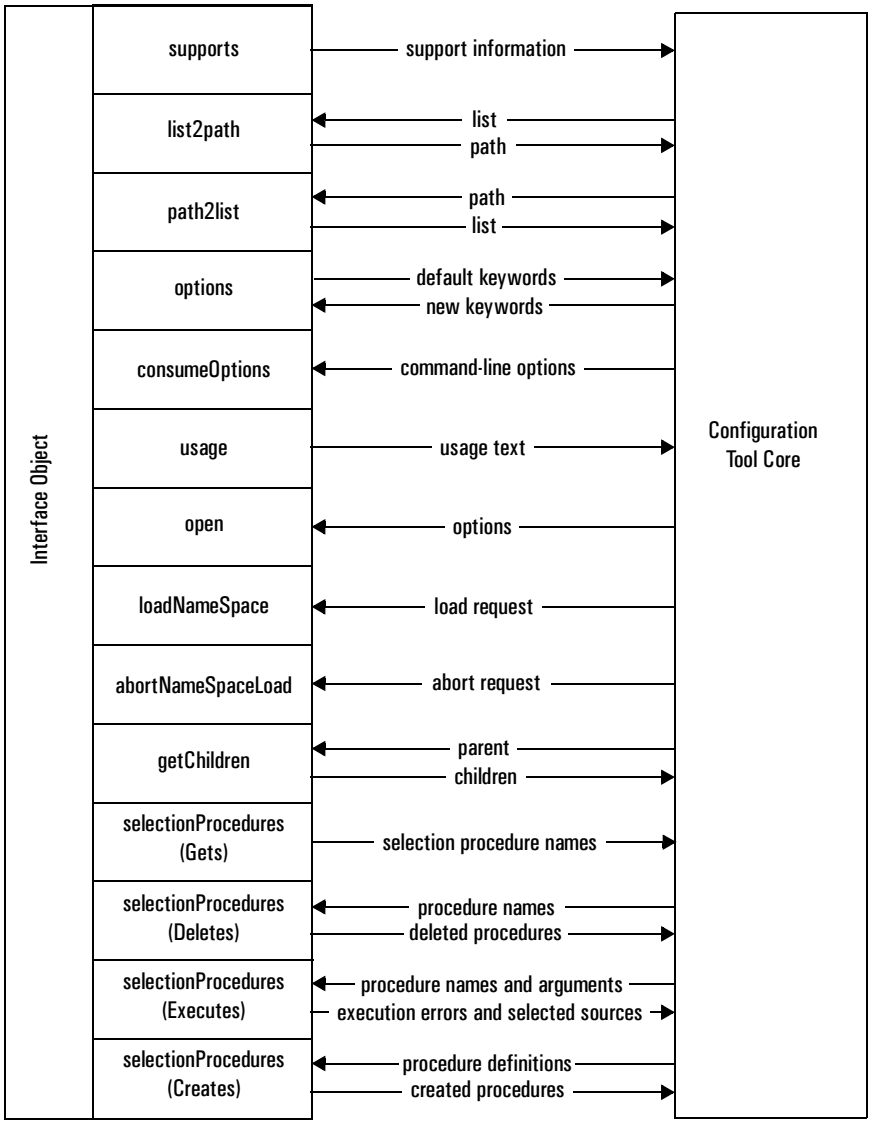
Interface Object Methods

The following methods are typically defined for interface objects:

Method Name	Description	Page
supports	Indicates whether or not a feature is supported.	4-10
list2path	Converts a Tcl list to a path string.	4-12
path2list	Converts a path string to a Tcl list.	4-13
options	Queries and sets command-line option keywords.	4-14
consumeOptions	Processes command-line arguments.	4-15
usage	Returns command-line usage information.	4-16
open	Loads configuration information and prepares the object for use.	4-17
loadNameSpace	Loads the target system's name space.	4-18
abortNameSpaceLoad	Aborts the current name-space load in progress.	4-20
getChildren	For the target system's name space, returns the child node names under a parent node as a Tcl list.	4-21
selectionProcedures	Deletes or creates selection procedures.	4-22

Many of these methods must be defined since they are invoked by the configuration tool. Others are invoked by the communication object and may be needed to create a working interface object.

The following illustration shows the interaction between the interface object methods and the configuration tool's core.



To develop a new configuration tool interface object, follow these steps:

Step 1: Develop an [incr Tcl] method to indicate which features are supported.

For example, if the target system type's short name is MYSYS, the method should have the following name:

```
elConfigMYSYSClass::supports
```

For a description of how to develop this method, see "Developing the supports Method" on page 4-10.

Step 2: Develop two [incr Tcl] methods to convert a Tcl list to a path string and to convert a path string to a Tcl list.

For example, if the target system type's short name is MYSYS, the methods should have the following names:

```
elConfigMYSYSClass::list2path
elConfigMYSYSClass::path2list
```

For a description of how to develop this method, see "Developing the list2path Method" on page 4-12 and "Developing the path2list Method" on page 4-13.

Step 3: Develop three [incr Tcl] methods to get and set command-line option keywords, parse command-line options, and return the text needed for a command-line usage message.

For example, if the target system type's short name is MYSYS, the methods should have the following names:

```
elConfigMYSYSClass::options
elConfigMYSYSClass::consumeOptions
elConfigMYSYSClass::usage
```

For a description of how to develop these methods, see "Developing the options Method" on page 4-14, "Developing the consumeOptions Method" on page 4-15, and "Developing the usage Method" on page 4-16.

Step 4: Develop an [incr Tcl] method to open a connection to the target system.

For example, if the target system type's short name is MYSYS, the method should have the following name:

```
elConfigMYSYSClass::open
```

For a description of how to develop this method, see "Developing the open Method" on page 4-17.

Step 5: Optionally develop two Tcl methods to load the target system's name space and to abort name space loading.

For example, if the target system type's short name is MYSYS, the methods should have the following names:

```
elConfigMYSYSClass::loadNameSpace
elConfigMYSYSClass::abortNameSpaceLoad
```

For a description of how to develop these methods, see "Developing the loadNameSpace Method" on page 4-18 and "Developing the abortNameSpaceLoad Method" on page 4-20.

Step 6: Develop an [incr Tcl] method to return the child node names in the target system's name space as a Tcl list.

For example, if the target system type's short name is MYSYS, the method should have the following name:

```
elConfigMYSYSClass::getChildren
```

For a description of how to develop this method, see "Developing the getChildren Method" on page 4-21.

Step 7: Optionally develop an [incr Tcl] method to delete or create user-configured selection procedures for the communication object.

For example, if the target system type's short name is MYSYS, the method should have the following name:

```
elConfigMYSYSClass::selectionProcedures
```

For a description of how to develop this method, see "Developing the selectionProcedures Method" on page 4-22.

Developing the supports Method

You must develop the **supports** [incr Tcl] method. This method indicates whether or not a specified functionality is supported. The configuration tool repeatedly invokes this method at startup to determine what functionality this communication object provides and what configuration tool facilities are needed. When the configuration tool invokes **supports**, it passes this method one parameter: a keyword identifying a unit of functionality that the configuration tool wants to know about. The expected return value is either 0 or 1. The **supports** method must recognize the following keywords:

receive data Indicates whether or not the communication object supports the flow of data from the target system to the data server. A return value of 0 disables the Edit Method window's Direction/system 1-to-system 2 radio button.

transmit data Indicates whether or not the communication object supports the flow of data from the data server to the target system. A return value of 0 disables the Edit Method window's Direction/system 2-to-system 1 radio button.

access Indicates whether or not the communication object supports the display and editing of access information. A return value of 1 adds an "Access" menu item to the main window's Edit menu.

trigger Indicates whether or not the communication object supports the display and editing of system-specific trigger information. A return value of 1 adds a system-specific trigger panel to the configuration tool's Trigger window.

trigger focus Indicates whether or not the communication object supports an Edit Mapping window focus policy. A return value of 1 allows a branch in the appropriate Edit Mapping window to be shown in a highlighted color.

receive spooling Indicates whether or not the data server's communication object supports the spooling of data received from the target system. A return value of 0 disables a spooling item on the main window's Edit/Spooling cascade menu.

transmit spooling Indicates whether or not the data server's communication object supports the spooling of data sent to the target system. A return value of 0 disables a spooling item on the main window's Edit/Spooling cascade menu.

dynamic name space Indicates whether or not the communication object supports a changing (dynamic) name space. A return value of 1 causes the configuration tool to check for new and deleted nodes when getting the children of a node.

name space loading Indicates whether or not the communication object supports the manual loading of name-space information. A return value of 1 adds a "Load Name-space" menu item to the main window's Control menu.

selection procedures Indicates whether or not the communication object supports dynamic mapping. A return value of 0 dims the Select checkbox in the Edit Mapping window.

All unrecognized keywords should cause the **supports** method to return 0, which means the unrecognized unit of functionality is not supported.

The following example of a **supports** method shows that the communication object can receive data from and transmit data to the target system, and can spool data sent to the target system.

Example

```
method supports args {

   set l [llength $args]

   # if `keyword' is omitted, return a list of
   # recognized keywords
   if {$l == 0} {
      return [list "receive data" "transmit data" "access" \
         "trigger" "trigger focus" "receive spooling" \
         "transmit spooling" "dynamic name space" \
         "name space loading" "selection procedures"]

   # else if `keyword' is present
   } elseif {$l == 1} {
      set k [lindex $args 0]
      switch $k {
               "receive data" {return 1}
              "transmit data" {return 1}
                     "access" {return 0}
                   "trigger" {return 0}
              "trigger focus" {return 0}
           "receive spooling" {return 0}
          "transmit spooling" {return 1}
        "dynamic name space" {return 0}
        "name space loading" {return 0}
      "selection procedures" {return 0}
                    default {if {$debug} \
                             {error "Unrecognized keyword: $k"}}
      }

   # else more than one argument has been supplied, error
   } else {
      utlArgEnd
   }

   return 0
}
```

Developing the list2path Method

You must develop the **list2path** [incr Tcl] method. This method converts a Tcl list into a path string specification. A similar method can optionally be developed for the data server (see "Developing the list2path Method" on page 3-16).

The syntax of the path string specification should be appropriate for the type of target system. For example, if the type of target system is Agilent Technolgies' RTAP product, path specifications would use the same syntax as RTAP's symbolic database addressing.

The configuration tool invokes the **list2path** method to convert Tcl lists to ordinary path string specifications. The configuration tool displays the paths to the end user as ordinary path strings but performs all computations on paths when they are encoded as Tcl lists.

The **list2path** method must implement exactly the opposite conversions that are implemented in the **path2list** method.

When the configuration tool invokes **list2path**, it passes this method one parameter: the Tcl list to be converted to a path string specification. It expects the **list2path** method to return the resulting path string specification.

The following example of a **list2path** method assumes that the path string syntax is similar to that used by the UNIX file system; that is, item names separated by solidus characters (/).

Example

```
method list2path p {
   return utlList2Path $p
}
```

Developing the path2list Method

You must develop the **path2list** [incr Tcl] method. This method converts a path specification into a Tcl list.

The configuration tool invokes **path2list** to convert ordinary path string specifications to paths expressed as Tcl lists. This method must implement exactly the opposite conversions that are implemented in the **list2path** method.

When the configuration tool invokes **path2list**, it passes this method one parameter: the path string to be converted to a Tcl list. It expects the **path2list** method to return the resulting Tcl list.

The following example of a **path2list** method assumes that the path string syntax is similar to that used by the UNIX file system; that is, item names separated by solidus characters (/).

Example

```
method path2list p {
   return utlPath2List $p
}
```

Developing the options Method

You must develop the **options** [incr Tcl] method. This method gets or sets the command-line option keywords supported by the communication object. A similar method can optionally be developed for the data server (see "Developing the options Method" on page 3-18).

When starting the configuration tool, you can type in command-line options on your operating system's command line; for example, **-file** <*name*>. Command-line option keywords, **-file** in this example, distinguish one command-line option from another.

The configuration tool invokes **options** up to two times during startup. The first invocation is to get the list of default option keywords supported by the communication object. If any of those keywords are also used by another communication object or by the configuration tool's core, this method is invoked a second time to rename those keywords.

When the configuration tool invokes **options** the first time, it does not pass in any parameters. The data server expects the **options** method to return a Tcl list of default command-line option keywords. When the configuration tool invokes **options** the second time, it passes in one parameter: a Tcl list with the new (possibly changed) names for the communication object's command-line option keywords. The data server expects the **options** method to return a Tcl list of these new command-line option keywords. The number of items that the configuration tool includes in this Tcl list will always equal the number of items that were returned by the configuration tool's first invocation of the **options** method.

The following example of an **options** method assumes that the communication object supports two command-line options: **-option1** *name* (an option with one argument) and **-option2** (an option with no arguments).

Example

```
method options {args} {
   if {[llength $args] > 0} {
      set optkeys [lindex $args 0]
      if {[llength $optkeys] == 2} {
         lassign $optkeys option1Keyword option2Keyword
      }
   }

   # Return the current command-line option keywords.
   return [list $option1Keyword option2Keyword]
}
```

Developing the consumeOptions Method

You must develop the **consumeOptions** [incr Tcl] method. This method parses and consumes the command-line options contained in a specified global variable. A similar method can optionally be developed for the data server (see "Developing the consumeOptions Method" on page 3-19).

The configuration tool invokes this method during startup to provide the communication object with an opportunity to parse its command-line options in the configuration tool command line.

At startup, the configuration tool invokes this method, passing it one parameter: the name of the global variable containing the program's command-line options as a Tcl list. This method should examine the list and remove any options that apply to the communication object. All unrecognized options should be left untouched since the configuration tool passes the resulting list to the other communication object, then consumes all command-line options supported by the core. If any command-line options remain after this, the configuration tool reports a usage error and exits.

To parse for and remove recognized command-line options, use the **utlGetArg** utility (see "utlGetArg, utlPeekArg, utlArgEnd" on page 8-43). You may use the **utlPeekArg** utility to examine, but not remove, a command-line option that the configuration tool's core supports and will eventually remove.

In the following example, the **consumeOptions** method parses and consumes the options supported in the example for the **options** method.

Example

```
method consumeOptions {varName} {
   upvar $varName args

      # Fetch command-line options.
      set opt1 [utlGetArg $option1Keyword {default value}]
      set opt2 [utlGetArg $option2Keyword]
   }
```

Developing the usage Method

You must develop the **usage** [incr Tcl] method. This method returns the text for a command-line usage message. The configuration tool invokes this method whenever command-line usage errors are detected. The configuration tool uses the information returned by this method to compose and print a human-readable usage message. A similar method can optionally be developed for the data server (see "Developing the usage Method" on page 3-20).

The returned value is a Tcl list with one entry per command-line option. Each entry is itself a list with two elements: the option string and an English explanation of the option. If the English explanation will not fit on one line, you may indent the text by inserting new-line characters and space characters. The configuration tool converts all new-line characters followed by space characters in the text into a single space character.

In the following example, the **usage** method returns the text in the example for the **options** method.

Example

```
method usage {} {
   return "
      {{\[$option1Keyword <name>]} {An option with a
         parameter. Feel free to run this description over
         multiple lines.}}
      {{\[$option2Keyword]}                {A flag option.}}"
}
```

Developing the open Method

You must develop the **open** [incr Tcl] method. This method establishes a connection between the configuration tool and the target system. A similar method can optionally be developed for the data server (see "Developing the open Method" on page 3-21).

The configuration tool invokes the **open** method during startup. When the configuration tool invokes **open**, it passes this method zero or more optional parameters. These parameters influence how the connection is opened.

- The **-configDir** <*name*> option sets the configuration directory to <name>, which is the full directory path name to the current object's configuration directory.
- The **-debug** option specifies that the communication object should print debug messages to stderr. This option can be useful to communication object authors for debugging.
- The **-stub** option specifies that the communication object should fake the connection to the target system. This can be useful for testing.
- The **-verbose** option specifies that the communication object should print status messages to stderr. This option can be useful to end users for debugging.

Example

```
method open args {

   global argv0 \
          env

      # fetch optional parameters
      set config [utlGetArg "-configDir" ""]
      set debug [utlGetArg "-debug"]
      set stub [utlGetArg "-stub"]
      set verbose [utlGetArg "-verbose"]
      utlArgEnd

      if {$debug} {puts stderr "DEBUG: $this open $args"}

      if {$verbose} {puts -nonewline stderr "[file tail $argv0]: \
         $this object connecting to xxx..."}

      < connect, or open a connection, to target system here >

          if {$verbose} {puts stderr " done"}
   }
```

Developing the loadNameSpace Method

You can optionally develop the **loadNameSpace** Tcl method. This method starts the transfer of name space data from the target system to the configuration tool, then returns without waiting for the transfer to complete. This name space data appears in one view of the configuration tool's Edit Mapping window.

If you don't need to explicitly load name space data, you can disable the name space loading functionality for the target system by setting the *name space loading* flag in the **supports** method to 0 (see "Developing the supports Method" on page 4-10). With this flag set to 0, you don't need to define this method and you may skip the rest of this section.

If you need to explicitly load name space data, you can enable name space loading functionality for the target system by setting the *name space loading* flag in the **supports** method to 1. With this flag set to 1, you must supply the configuration tool with a **loadNameSpace** method. The configuration tool invokes this method when the end user chooses the Load Name Space menu item from the main window's Control menu.

The configuration tool invokes this method whenever it needs to initiate a name space load operation. The configuration tool passes two optional parameters to this method. The optional parameter **-command** <*cmd*> specifies the Tcl command to execute after the transfer is complete. The optional parameter **-statusCommand** <*scmd*> specifies the Tcl command to periodically execute while the transfer is in progress, which updates the Status window.

For the **-statusCommand** option, all occurrences of %message in the Tcl script <*scmd*> must be replaced by a status message just prior to the execution of <*scmd*>.

For the **-command** option, all occurrences of %error in the Tcl script <*cmd*> must be replaced by 0 if no errors occurred and 1 if errors occurred, just prior to the execution of <*cmd*>. If errors occurred, all occurrences of %message in <*cmd*> must be replaced by the error message text just prior to the execution of <*cmd*>.

Example

```
method loadNameSpace args {

   # fetch optional parameters
   set  cmd [utlGetArg "-command"       ""]
   set scmd [utlGetArg "-statusCommand" ""]
   utlArgEnd

   if {$debug} {puts stderr "DEBUG: $this loadNameSpace $args"}
   < start background load here >
}

# the background name space loading code should run this
# code periodically
proc loadNameSpaceStatus {} {
   if {< an error occurs here > && ![utlIsNull $cmd]} {
      set error 1
      set message < my error message here >

      # replace all occurrences of %error and %message
      regsub -all "%error" $cmd \
         [utlQuote [utlQuote -backSlash $error]] rcmd
      regsub -all "%message" $rcmd \
         [utlQuote [utlQuote -backSlash $message]] rcmd
      uplevel #0 $rcmd
   }

   if {![utlIsNull $scmd]} {
      set message < my status message here >

      regsub -all "%message" $scmd \
         [utlQuote [utlQuote -backSlash $message]] rcmd
      uplevel #0 $rcmd
   }
}

# when the background name space loading code is done,
# it should run this code
proc loadNameSpaceDone {} {
   if {![utlIsNull $cmd]} {
      set error 0
      set message ""

         # replace all occurrences of %error and %message
         regsub -all "%error" $cmd \
            [utlQuote [utlQuote -backSlash $error]] rcmd
         regsub -all "%message" $rcmd \
            [utlQuote [utlQuote -backSlash $message]] rcmd
         uplevel #0 $rcmd
      }

}
```

Developing the abortNameSpaceLoad Method

You can optionally develop the **abortNameSpaceLoad** Tcl method. This method aborts the transfer of name space data previously started by the **loadNameSpace** method.

If the **loadNameSpace** method is defined, this method must also be defined. If the **loadNameSpace** method is not defined, you do not need to define this method.

The configuration tool invokes the **abortNameSpaceLoad** method when the end user clicks on the Cancel button in the Name Space Load Status window. The configuration tool automatically created the Name Space Load Status window when it invoked the **loadNameSpace** method.

Example

```
method abortNameSpaceLoad {} {
   if {$debug} {puts stderr "DEBUG: $this abortNameSpaceLoad"}
   < abort name-space load operation here >

}
```

Developing the getChildren Method

You must develop the **getChildren** [incr Tcl] method. This method returns the names of children under a specified node for the target system's name space.

The configuration tool invokes this method whenever it needs to get the names of children nodes. When the configuration tool invokes **getChildren**, it passes this method the full path name of the node (as a Tcl list) whose child names are wanted. This method should return a Tcl list of child names that appear directly under this node. The returned child names should not include any path information.

The optional parameter **-filter** <*pattern*> specifies a pattern that filters child names. Only child names that match <*pattern*> should be returned. If this parameter is omitted, all child names should be returned.

By definition, the root node of the name space tree is the empty string.

The configuration tool assumes that the target system's name space can be represented with a tree structure. For target systems with a flat, linear address space, simply make all the system's data items children of the root node. The end user can use the configuration tool's built-in filtering facilities to make this list manageable.

Example

```
method getChildren args {

   # fetch optional parameter
   set fil [utlGetArg "-filter" "*"]

   # fetch required parameter
   set n [utlGetArg]
   utlArgEnd

   if {$debug} {puts stderr "DEBUG: $this getChildren $args"}

   return < compose a Tcl list of the nodes under `$n' matching
          `$fil' here >

   }
```

Developing the selectionProcedures Method

If you want your communication object to support dynamic mapping, you must develop the **selectionProcedures** [incr Tcl] method. This method gets, deletes, executes or creates selection procedures. The **selectionProcedures** method has four modes of operation.

1. Gets

In its first mode, **selectionProcedures** takes no arguments and returns a list of selection procedures supported by the source communication object, which includes selection procedures originating in both the communication object and the data server.

The returned list contains three elements: the procedure name, procedure arguments, and the application programming interface (API) version of the selection procedure. For the E.02.20 version of Enterprise Link, the API version should always be 1.

Example

```
if {[llength $args] == 0} {
   set rc {}
   foreach proc_name [info procs {xxx_select_*}] {

         regsub "^xxx_select_(.*$)" $proc_name {\1} p_name

         # --- fetch procedure's API version number ---
      if {[info exists select_api_versions($p_name)]} {
         set p_version $select_api_versions($p_name)
      } else {
         set p_version 0  ;# version is unknown
      }

         # --- properly convert default selection procedure arguments ---
         set p_args {}
         foreach arg_name [info args $proc_name] {
            if {[info default $proc_name $arg_name arg_value]} {
               lappend proc_args [list $arg_name $arg_value]
            } else {
               lappend proc_args $arg_name
            }
         }

      lappend rc [list $p_name $p_args $p_version]
   }

   return $rc
}
```

2. Deletes

In its second mode of operation, the **selectionProcedures** method takes any number of selection procedure names as arguments and returns a Tcl list of the names of deleted procedures.

Example

```
set rc {}
foreach arg $args {

   set arg_length [llength $arg]

   if {$arg_length == 1} {
      set p_name [lindex $arg 0]

      rename xxx_select_$p_name {}

      unset select_api_versions($p_name)
      if {[array size select_api_versions] == 0} {
         unset select_api_versions
      }

      lappend rc $p_name
   }
}
```

3. Executes

In its third mode of operation, the **selectionProcedures** method takes any number of two-item arguments. The first list item is a selection procedure name and the second is a list of that selection procedure's arguments.

This third mode of **selectionProcedures** returns a Tcl list of zero or more three-item lists. The first item is the selection procedure executed, along with the selection procedure's arguments. The second item is the error code generated from the execution. The third item is a list of selected source paths, with each path in the form of a Tcl list. If the procedure execution is successful, the item in the list representing the error code will be empty and the last item will contain selected source paths. If the procedure execution was unsuccessful, the error code will appear in the list and the following item representing source paths will be empty.

Example

```
set rc {}
foreach arg $args {

   set arg_length [llength $arg]

   # --- just an execute-procedure request? ---
   if {$arg_length == 2} {
      set p_name [lindex $arg 0]
      set p_args [lindex $arg 1]

      # --- set defined arguments ---
      set el_args [list $this]

      set sources {}

      if {[info exists select_api_versions($p_name)]} {

         # --- if no errors occur, "err_code" item is empty ---
         set err_code {}

         set api_version $select_api_versions($p_name)

         # --- invoke selection procedure appropriately ---
         switch -- $api_version {
            "0" -
            "1" {if {[catch {set sources [eval xxx_select_$p_name \
            $el_args $p_args]}]} {
                  global errorInfo errorCode
                  set err_code [list $errorInfo $errorCode]
                 }
         }

         default {set err_code [utlNls "API version mismatch error: \
         selection-procedure `%0' requires unsupported API version #%1" \
         $p_name api_version]
         }

      } else {
         set err_code [utlNls "Undefined selection-procedure \
         (procedure=%0)" $p_name]
      }

      lappend rc [list $arg $err_code $sources]
   }
}
```

4. Creates

In its fourth mode, the **selectionProcedures** method takes any number of arguments in a four-item list. The first item is a selection procedure name. The second item is a list of that selection procedure's arguments. The third item is the body text defining the selection procedure, and the fourth item is the application programming interface (API) version of the selection procedure. In the E.02.20 version of Enterprise Link, the API version should always be 1.

This fourth mode of the **selectionProcedures** method returns a Tcl list of zero or more procedure names.

Example

```
set rc {}
foreach arg $args {

   set arg_length [llength $arg]

   if {$arg_length == 4} {
      set p_name [lindex $arg 0]
      set p_args [lindex $arg 1]
      set p_body [lindex $arg 2]
      set p_vers [lindex $arg 3]

      # --is this an unsupported version of the selection-procedure API? --
      if {$p_vers > 1} {
         error [utlNls "Unsupported version of the selection-procedure \
         API (procedure=%0 version=%1)" $p_name $p_vers]
      }

      proc xxx_select_$p_name $p_args $p_body
      set select_api_versions($p_name) $p_vers
      lappend rc $p_name
   }
   return $rc
}
```

Creating a Message Catalog File

To support localization, the configuration tool supports the concept of message catalog files. These files allow you to easily change displayed text.

Message catalog files are simply Tcl scripts stored under the directory <environment variable ELROOT>/lib/nls/msg/C

These Tcl scripts assign character strings to elements of the global Tcl array *utl_msg_cat*. This array is indexed by either a default string or a message catalog message ID. Short character strings index into the *utl_msg_cat* array with a default string, while very long character strings use a catalog message ID.

Rather than accessing this array directly, internationalized Tcl programs use the **utlNls** utility to look up a message (see "utlNls" on page 8-54). This utility is passed either the default string or catalog message ID, and zero or more optional parameters. This utility returns the localized and formatted string.

If the **utlNls** utility cannot find the specified string in the global *utl_msg_cat* array, this utility simply returns the supplied default string or message catalog ID (as a string). This utility also provides limited string formatting capabilities.

Before Tcl message catalog files can be used, they must be loaded. A Tcl message catalog file is loaded by sourcing it. All Tcl message catalog files reside in the directory specified by the global Tcl variable *el_app_msg_cat_dir*. A Tcl message catalog file should only be loaded if this global Tcl variable exists. The following example shows how to load a Tcl message catalog file. The Tcl code shown in this example should appear near the beginning of the first file that uses the messages defined in this message catalog file.

Example

```
# load message catalog file
if {[info exists el_app_msg_cat_dir]} {
   source ${el_app_msg_cat_dir}/my.msgcat
}
```

5

Developing an Access Window

Developing an Access Window

This chapter describes how to develop the optional target-system access window for the Agilent Technologies Enterprise Link configuration tool.

The configuration tool uses access windows to provide the end user with the ability to specify system-specific access information. This includes connection and startup information that is unique to the target system.

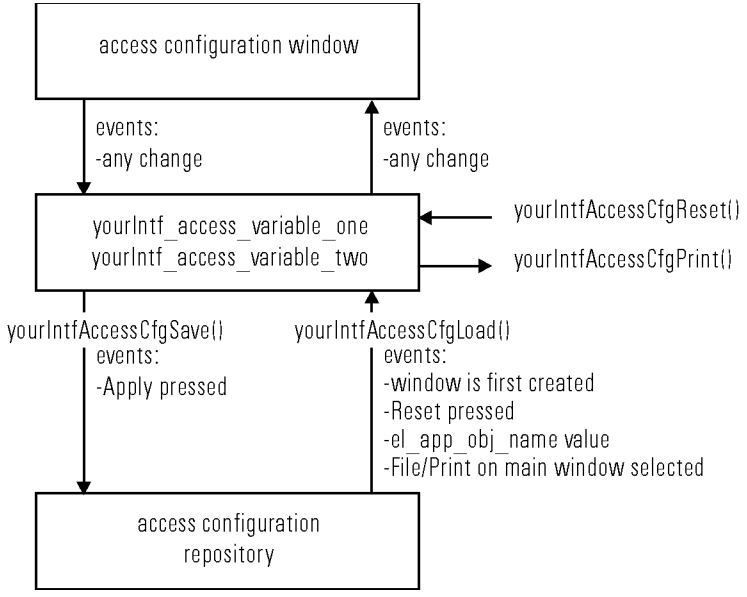
Access windows are optional. If the target system does not have any system-specific access information, you can disable access window functionality for the target system by setting the *Access* flag in the **supports** method to 0 (see "yourIntfClass::supports" on page 7-43). With this flag set to 0, you do not need to define any access window procedures for the target system, and you may skip the rest of this chapter.

If the system needs system-specific access information, you must enable access window functionality for the target system by setting the `Access` flag in the **supports** method to 1. When this flag is set to 1, you must supply the configuration tool with a collection of access window procedures. When the configuration tool is running, it will call these access window procedures as needed.

The configuration tool supports up to one access window for each type of target system. Access configuration data is maintained on a per configuration-object basis.

Access Window Data Flow

The following illustration shows how data flows between the access window's Tk widgets, data variables that you define, and configuration repository. In this illustration, the target-system type's short name is *yourIntf*.



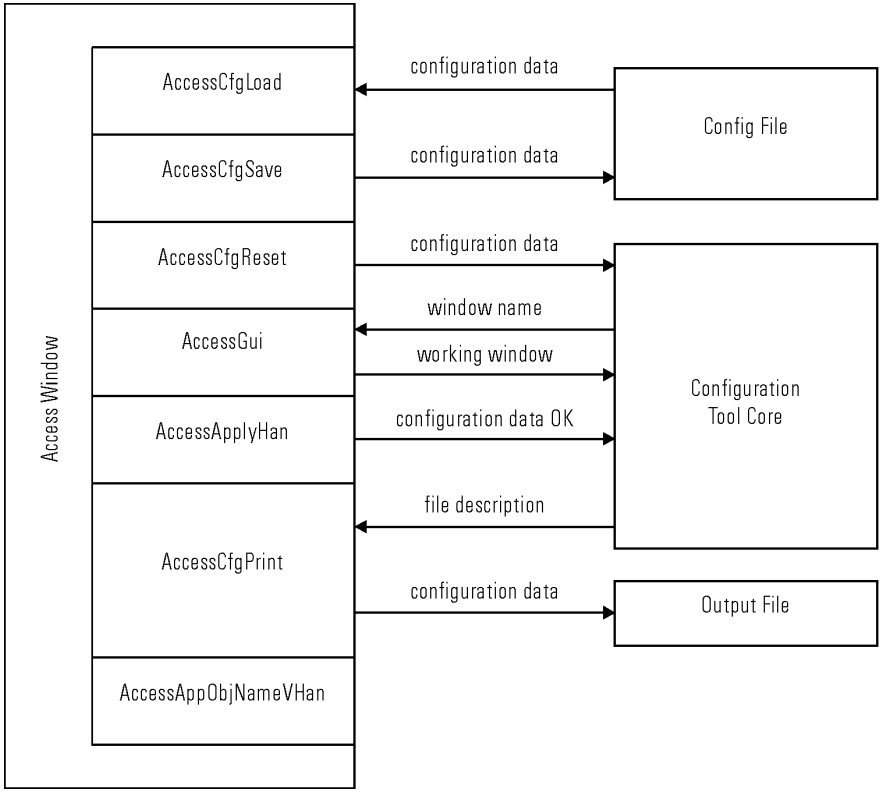
Access Window Procedures

The following procedures are typically defined for access windows.

Procedure Name	Description	Page
yourIntfAccessCfgRest	Creates and initializes access window variables.	5-8
yourIntfAccessCfgLoad	Loads access window configuration data.	5-9
yourIntfAccessCfgSave	Saves access window configuration data.	5-11
yourIntfAccessCfgPrint	Prints access window configuration data to a file.	5-13
yourIntfAccessGui	Creates and displays the access window.	5-15
yourIntfAccessApplyHan	Handles user clicks of the access window's Apply button.	5-19
yourIntfAccessAppObjNameVHan	Handles when the currently open configured object changes.	5-21

Some of these procedures must be defined since they are called by the configuration tool. Others are called by the communication object and are needed to create a working access window.

The following illustration shows the interaction between the access window procedures, configuration file, configuration tool's core, and output file.



To develop a system-specific access window, follow these steps:

Step 1: Identify and define the connection and startup information required for the type of target system.

For example, this may include a host computer name, a user login name, and a user password.

Step 2: Assign a variable name to each of the items identified in the previous step.

To avoid name collisions with variables defined by the configuration tool and its associated libraries, prefix each name with the target-system type's short name. Also, to avoid name collisions with other variables associated with the target-system type, follow the short name prefix with the string *_access*.

For example, if the target-system type's short name is MYSYS and if two variables are required, the variables should have the following names:

```
MYSYS_access_variable_one
MYSYS_access_variable_two
```

Step 3: Develop four Tcl procedures to initialize, load, store, and print the data defined in the previous step.

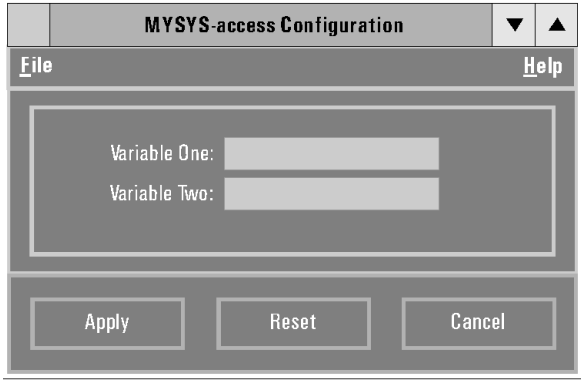
For example, if the target-system type's short name is MYSYS, the procedures should have the following names:

```
MYSYSAccessCfgReset
MYSYSAccessCfgLoad
MYSYSAccessCfgSave
MYSYSAccessCfgPrint
```

For a description of how to develop these procedures, see
"Developing the yourIntfAccessCfgReset Procedure" on page 5-8,
"Developing the yourIntfAccessCfgLoad Procedure" on page 5-9,
"Developing the yourIntfAccessCfgSave Procedure" on page 5-11, and
"Developing the yourIntfAccessCfgPrint Procedure" on page 5-13.

Step 4: Lay out the access window.

The following shows a sample access window built with Tk widgets.



Step 5: Develop a Tcl procedure to create and display the access window.

For example, if the target system type's short name is MYSYS, the procedure should have the following name:

```
MYSYSAccessGui
```

For a description of how to develop this procedure, see "Developing the yourIntfAccessGui Procedure" on page 5-15.

Step 6: Develop Tcl procedures to handle when the end user clicks on the access window's Apply button or when the global variable *el_app_obj_name* changes value.

For example, if the target-system type's short name is MYSYS, the procedures should have the following names:

```
MYSYSAccessApplyHan
MYSYSAccessAppObjNameVHan
```

For a description of how to develop these procedures, see "Developing the yourIntfAccessApplyHan Procedure" on page 5-19 and "Developing the yourIntfAccessAppObj- NameVHan Procedure" on page 5-21.

Developing the yourIntfAccessCfgReset Procedure

You must develop the **yourIntfAccessCfgReset** Tcl procedure. This procedure creates then sets the access window's variables to their default state. These variables must be global for Tk's widgets to be able to access them.

The configuration tool calls this procedure during startup to create and initialize the global variables needed by the target system's access window.

In addition to creating the access window's data variables that you defined, this procedure must create and initialize a configuration file revision variable. The value of this variable represents the file format used for access configuration data that is stored in the configuration repository. This variable will be saved whenever the access window's configuration is saved, and checked whenever the access window's configuration is loaded.

The following example of an **yourIntfAccessCfgReset** procedure assumes the access window requires two variables: *yourIntf_access_variable_one* and *yourIntf_access_variable_two*.

Example

```
proc yourIntfAccessCfgReset {} {
 global yourIntf_access_config_app_rev \
        yourIntf_access_variable_one \
        yourIntf_access_variable_two

   # set access config-file-format revision number
   set yourIntf_access_config_app_rev "1"

   # initialize window variables
   set yourIntf_access_variable_one <default value>
   set yourIntf_access_variable_two <default value>
}
```

Developing the yourIntfAccessCfgLoad Procedure

You must develop the **yourIntfAccessCfgLoad** Tcl procedure. This procedure loads the access configuration stored in the access configuration repository. The configuration data is stored in a file name specified by the global Tcl variable *yourIntf_access_cfg_file_name*. This file resides in the base directory specified by the global Tcl variable *el_app_obj_name* under the subdirectory specified by the global Tcl variable *el_app_cfg_subdir*. All of these global variables are defined and set by the core component of the configuration tool.

The configuration tool does not directly call this procedure. It is called indirectly when the configuration tool calls **yourIntfAccessGui** and **yourIntfAccessCfgPrint**. It is also called when the end user clicks on the access window's Reset button. These causal relationships are set up in the **yourIntfAccessGui** procedure.

This procedure must be able to read the configuration file written by **yourIntfAccessCfgSave**. Since **yourIntfAccessCfgSave** writes the configuration data to the repository in the form of a Tcl script, all this procedure has to do is source that script.

In case the configuration file does not exist, first call **yourIntfAccessCfgReset** to provide a default access configuration before calling **yourIntfAccessCfgLoad**. This also ensures that the access window's Reset button properly resets the displayed window data when there is no configuration file.

To verify that the contents of the just-read configuration file are valid, call the **utlChkCfgFileRev** utility (see "utlChkCfgFileRev" on page 8-21).

The following example of an **yourIntfAccessCfgLoad** procedure assumes the access window requires two variables: *yourIntf_access_variable_one* and *yourIntf_access_variable_two*.

Example

```
proc yourIntfAccessCfgLoad {} {
 global el_app_cfg_subdir \
        el_app_obj_name \
        yourIntf_access_cfg_file_name \
        yourIntf_access_config_app_rev \
        yourIntf_access_variable_one \
        yourIntf_access_variable_two

   # compose file name
   set fname "${el_app_obj_name}${el_app_cfg_subdir}/\
   $yourIntf_access_cfg_file_name"

   # if the specified config file does not exist or is empty
   if {([file exists $fname] != 1) || ([file size $fname] == 0)} {
      # use the default values
      yourIntfAccessCfgReset
      return
   }

   # load the access configuration
   set _prefix "yourIntf_access_"
   source $fname

   utlChkCfgFileRev "yourIntf_access_config_rev" \
      $yourIntf_access_config_app_rev $fname
}
```

Developing the yourIntfAccessCfgSave Procedure

You must develop the **yourIntfAccessCfgSave** Tcl procedure. This procedure saves the access configuration to the access configuration repository. The configuration data is stored in a file name specified by the global Tcl variable *yourIntf_access_cfg_file_name*. This file resides in the base directory specified by the global Tcl variable *el_app_obj_name* under the subdirectory specified by the global Tcl variable *el_app_cfg_subdir*. All of these global variables are defined and set by the core component of the configuration tool.

The configuration tool does not directly call this procedure. It is called when the end user clicks on the access window's Apply button. This causal relationship is set up in the **yourIntfAccessGui** procedure.

This procedure must write a configuration file that can be read by the **yourIntfAccessCfgLoad** procedure. This procedure normally writes the configuration data in the form of a Tcl script.

Open the configuration file using the **utlOpen** utility and close using the **utlClose** utility (see "utlOpen, utlClose" on page 8-55). These utilities provide transparent support for configuration file versioning and safe file writes.

The following shows the usual layout of an access configuration file:

Example

```
# "Enterprise Link yourIntf-access Configuration File"

if {![info exists _prefix]} {set _prefix "yourIntf_Access_"}

set ${_prefix}variables "${_prefix}config_rev ${_prefix}<name1> \
    ${_prefix}<name2> ..."

set ${_prefix}config_rev 1
set ${_prefix}<name1> {<my value1>}
set ${_prefix}<name2> {<my value2>}
:          :                  :
:          :                  :

# EOF
```

where

_prefix Allows configuration file loading procedures to attach a custom prefix string to the variable names loaded.

<name#> The name of an access window variable.

<my value#> The current value of an access window variable.

The following example of an **yourIntfAccessCfgSave** procedure assumes the access window requires two variables: *yourIntf_access_variable_one* and *yourIntf_access_variable_two*.

Example

```
proc yourIntfAccessCfgSave {} {
 global el_app_cfg_subdir \
        el_app_obj_name \
        el_prod_name \
        yourIntf_access_cfg_file_name \
        yourIntf_access_config_app_rev \
        yourIntf_access_variable_one \
        yourIntf_access_variable_two

   # compose file name
   set fname "${el_app_obj_name}${el_app_cfg_subdir}/\
   $yourIntf_access_cfg_file_name"

   # open the output file
   set f [utlOpen $fname w]

   puts $f "# $el_prod_name yourIntf-access Configuration File"
   puts $f ""
   puts $f "if {!\[info exists _prefix\]} \
      {set _prefix \"yourIntf_access_\"}"
   puts $f ""
   puts $f "set \${_prefix}variables \"\${_prefix}config_rev \
      \${_prefix}variable_one \${_prefix}variable_two\""
   puts $f ""

   # save the access configuration
   puts $f "set \${_prefix}config_rev \
      [list $yourIntf_access_config_app_rev]"
   puts $f "set \${_prefix}variable_one \
      [list $yourIntf_access_variable_one]"
   puts $f "set \${_prefix}variable_two \
      [list $yourIntf_access_variable_two]"
   puts $f ""

   puts $f "# EOF"

   utlClose $f
}
```

Developing the yourIntfAccessCfgPrint Procedure

You must develop the **yourIntfAccessCfgPrint** Tcl procedure. This procedure writes human-readable documentation about the current access configuration to a specified file descriptor. The written text should be easy to understand and complete.

The configuration tool calls this procedure whenever the end user chooses the Print menu item from the File menu in the configuration tool's main window.

When the configuration tool calls this procedure, the procedure passes to the configuration tool one parameter: an open file descriptor that all output should be written to. This procedure should only write to this file descriptor, it should not read from it nor close it.

To construct what is written, use the table formatting utilities **utlTableHeader**, **utlTableRow**, and **utlTablePut** (see "utlTableHeader, utlTableRow, utlTablePut" on page 8-71). With these utilities, the final output should be a table that looks something like the following:

yourIntf-access Configuration	
Item	Value
< name1 > < name2 > < name3 > :	< value1 > < value2 > < value3 > :

where

`<name#>` The name of an access window variable.

`<value#>` The current value of an access window variable.

Empty lines, as shown in the example table above, may be inserted to separate related configuration parameters. This can improve the generated documentation's readability, especially for tables with many rows.

The following example of an **yourIntfAccessCfgPrint** procedure assumes that the access window requires two variables: *yourIntf_access_variable_one* and *yourIntf_access_variable_two*.

Example

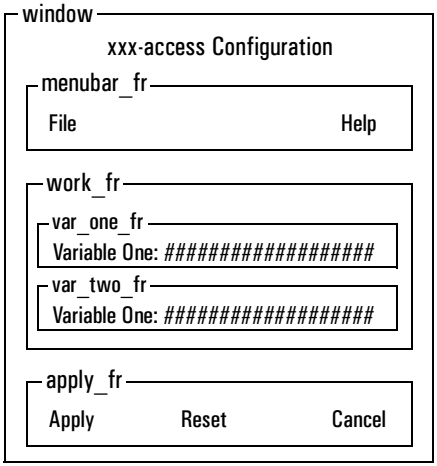
```
proc yourIntfAccessCfgPrint f {
 global yourIntf_access_variable_one \
        yourIntf_access_variable_two

   # load yourIntf-access configuration
   yourIntfAccessCfgLoad

   # generate yourIntf-access documentation
   utlTableHeader $f -title "yourIntf-access Configuration" \
      "Item" "Value"

   utlTableRow $f "Var One" $yourIntf_access_variable_one
   utlTableRow $f "Var Two" $yourIntf_access_variable_two
   utlTableRow $f

   utlTablePut $f -indent 2 -justify {right left}
}
```

Developing the yourIntfAccessGui Procedure

You must develop the **yourIntfAccessGui** Tcl procedure. This procedure creates, configures, and displays the access window.

The configuration tool calls **yourIntfAccessGui** whenever the end user chooses the yourIntf Access... menu item from the Edit menu in the configuration tool's main window.

Before you can write the **yourIntfAccessGui** procedure, you must lay out the access window's interface. Once you have done this, you must prepare your layout for use with Tk. The window's labels will be built using Tk label widgets. The window's text entry widgets will be built using Tk entry widgets. The window's buttons will be built using Tk button widgets. All of these widgets must be grouped together using Tk frame widgets. Due to the way the Tk pack command works, each frame widget must enclose only rows or columns of Tk widgets. The following figure shows a sample layout for access window frames.



When the configuration tool calls this procedure, it passes in one parameter: the name to use for the access window's top-level window. If **yourIntfAccessGui** finds that this window already exists, it must simply make the window visible by calling the **utlMkPanelVisible** utility (see "utlMkPanelPrologue, utlMkPanelEpilogue, utlMkPanelVisible" on page 8-52).

Since creating a window is relatively slow, call the **utlBusyCursor** utility before actually creating the window and the **utlIdleCursor** utility afterwards (see "utlBusyCursor, utlIdleCursor" on page 8-17). These utilities temporarily display the end user's mouse cursor as an hourglass sprite.

To create and configure the window's top-level window, call the **utlMkPanelPrologue** utility. After calling this utility, the variable *$self* is defined and the variable *$p* is modified. The *$self* variable is the handle of the newly created top-level window, while the *$p* variable is the prefix required for all of the window's widgets.

To conclude the creation of the top-level window, call the **utlMkPanelEpilogue** utility. This utility makes the window resizable, then positions and displays it.

Note that the **utlMkPanelPrologue** and **utlMkPanelEpilogue** utilities ensure that if anything goes wrong during window construction, that all widgets associated with the partially created window are automatically destroyed.

To process all of the window's widgets, call the **utlPrepareWidget** utility (see "utlPrepareWidget" on page 8-60). This utility ensures that the appearance and behavior of all widgets in the configuration tool are consistent. For entry and text widgets that are output only, use the utility's **-noEntry** option to ensure the widget's background color is appropriately set. When a check box or radio button widget controls whether or not an entry widget is enabled, use the utility's **-autoSelect** <*list*> option to ensure that the check box and radio button widgets automatically become enabled when the end user clicks a mouse button on the entry widget. The <*list*> option argument must be a Tcl list of all check box and radio button widgets that affect the state of the entry widget.

To arrange for focus traversal of the window's entry widgets, call the **utlFocusTraversal** utility (see "utlFocusTraversal" on page 8-39). Focus traversal occurs when an end user first selects a text-entry widget, then types the **<Return>** key. Typing the **<Return>** key causes the next entry widget to be selected for end user input.

The following example of a **yourIntfAccessGui** procedure assumes that the access window requires two variables: *yourIntf_access_variable_one* and *yourIntf_access_variable_two*.

Example

```
proc yourIntfAccessGui p {
 global el_app_obj_name \
        yourIntf_access_unotes_file_name \
        yourIntf_access_variable_one \
        yourIntf_access_variable_two

   utlBusyCursor

   # if the window already exists and has children
   if {[utlIsPanel $p]}{
      utlMkPanelVisible $p
      utlIdleCursor
      return
   }

   # prepare top-level window
   utlMkPanelPrologue p -title "yourIntf-access Configuration" \
      -class Access

   # create frame widgets
   frame ${p}menubar_fr -relief raised
   frame ${p}work_fr
   frame ${p}var_one_fr
   frame ${p}var_two_fr
   frame ${p}apply_fr
   utlPrepareWidget ${p}menubar_fr ${p}work_fr ${p}var_one_fr \
      ${p}var_two_fr {p}apply_fr

   < create the menubar and pull-down menu widgets here >

   # create variable one widgets
   label ${p}var_one_lb -text "Variable One:" -anchor e
   entry ${p}var_one_en -width 15 \
      -textvariable yourIntf_access_variable_one
   utlPrepareWidget ${p}var_one_lb ${p}var_one_en

   # create variable two widgets
   label ${p}var_two_lb -text "Variable Two:" -anchor e
   entry ${p}var_two_en -width 15 \
      -textvariable yourIntf_access_variable_two
   utlPrepareWidget ${p}var_two_lb ${p}var_two_en

   # setup entry widget traversal chain
   utlFocusTraversal ${p}var_one_en ${p}var_two_en

   # create Apply/Reset/Cancel push button widgets
   button ${p}apply_bt -text "Apply" \
      -command [list yourIntfAccessApplyHan $self]
   button ${p}reset_bt -text "Reset" \
      -command [list yourIntfAccessCfgLoad]
   button ${p}cancel_bt -text "Cancel" \
      -command [list destroy $self]
   utlPrepareWidget ${p}apply_bt ${p}reset_bt ${p}cancel_bt

   # load yourIntf-access config data into window
   if {[catch {yourIntfAccessCfgLoad}]} {
      global errorInfo errorCode
      destroy $self
      utlIdleCursor
      return -code error -errorinfo $errorInfo \
         -errorcode $errorCode $em
   }

   # arrange for this window to update whenever the variable
   # "el_app_obj_name" changes
   trace variable el_app_obj_name w \
         [list yourIntfAccessAppObjNameVHan $self]
   bind $self <Destroy> "
      if {\"%W\" == \"$self\"} {
```

```
         trace vdelete el_app_obj_name w \
            {[list yourIntfAccessAppObjNameVHan $self]}
         bind $self <Destroy> {}
      }
   "

   < pack widgets together here >

   # setup menubar
   focus ${p}menubar_fr

   # publish window
   utlMkPanelEpilogue $self

   utlIdleCursor
}
```

Developing the yourIntfAccessApplyHan Procedure

You must develop the **yourIntfAccessApplyHan** Tcl procedure. This procedure checks and then saves the current values of the access configuration variables to the access configuration repository.

If any invalid access configuration variable values are found, this procedure reports the error and then returns. In this case, the current values of the access configuration variables are not saved and the access window remains mapped, allowing the end user to fix the reported problem. Otherwise, if all values are found to be valid, the current configuration is saved.

The configuration tool never calls this procedure. Rather, it is called by Tk whenever the end user clicks on the access window's Apply button. This causal relationship is set up in the **yourIntfAccessGui** procedure. The **yourIntfAccessGui** procedure arranges for the name of the access window's top-level window to be passed as the first parameter to **yourIntfAccessApplyHan** when the causal relationship is set up.

When entry widgets have an associated enable/disable check box widget and the check box widget is disabled, do not check the current values of the dependent entry widgets. In this situation, illegal entry widget values are allowed.

Before saving the end user's changes, check for these common problems:

- Empty entry widgets.
- Numeric entry widgets filled with non-numeric text.
- Non-numeric entry widgets filled only with numeric text.
- Invalid combinations of enabled check box widgets.

To detect and report the first and third problem, use the **utlChkAlpha** utility (see "utlChkAlpha" on page 8-20). To detect and report the first and second problem, use the **utlChkInt** utility (see "utlChkInt" on page 8-23). You'll have to detect the last problem yourself and use the Tcl error command to report the problem to the end user. Be aware that the configuration tool intercepts errors and displays the error in an Error window.

Once you've verified that the current values of the access variables are correct, save them to the configuration file using **yourIntfAccessCfgSave**.

Since writing to files is relatively slow, call the **utlBusyCursor** utility before starting the write and the **utlIdleCursor** utility after completing it (see "utlBusyCursor, utlIdleCursor" on page 8-17). These utilities temporarily display the end user's mouse cursor as an hourglass sprite.

The following example of a **yourIntfAccessApplyHan** procedure assumes that the access window requires two variables: *yourIntf_access_variable_one* and *yourIntf_access_variable_two*.

Example

```
proc yourIntfAccessApplyHan p {
 global yourIntf_access_variable_one \
        yourIntf_access_variable_two

   if {![string match *. $p]} {set p "${p}."}

   # check fields...
   utlChkAlpha $yourIntf_access_variable_one \
      -comment "Expected variable name." \
      [utlWidgetText ${p}var_one_lb]
   utlChkAlpha $yourIntf_access_variable_one \
      -comment "Expected variable name." \
      [utlWidgetText ${p}var_two_lb]

   # save the now-validated access configuration
   utlBusyCursor
   yourIntfAccessCfgSave
   utlIdleCursor
}
```

Developing the yourIntfAccessAppObj-NameVHan Procedure

You must develop the **yourIntfAccessAppObjNameVHan** Tcl procedure. This procedure handles changes to the value of the global variable *el_app_obj_name*. The global variable *el_app_obj_name* contains the name of the object currently open for editing. This procedure reloads the access window with the access configuration associated with the new object.

This procedure is only called when the access window is on display and there is a change in the object currently open for editing.

The configuration tool does not directly call this procedure. Rather, it is called by Tcl whenever the end user chooses the Open... menu item from the File menu in the configuration tool's main window and opens an object for editing. This causal relationship is set up in the **yourIntfAccessGui** procedure. The **yourIntfAccessGui** procedure arranges for the name of the access window's top-level window to be passed as the first parameter to this procedure when the causal relationship is set up. Tcl passes this procedure three more parameters as part of its variable trace handling functionality.

Example

```
proc yourIntfAccessAppObjNameVHan {p name el op} {

   # if yourIntf-access configuration window does not exist,
   # bail out
   if {![utlIsPanel $p]} {return}

   yourIntfAccessCfgLoad
}
```

6

Developing a Trigger Panel

Developing a Trigger Panel

This chapter describes how to develop the optional system-specific trigger panel for the Agilent Technologies Enterprise Link configuration tool.

The configuration tool uses trigger panels to provide the end user with the ability to specify system-specific trigger information. This includes trigger sources and filters that are unique to the target system. Trigger panels are part of the configuration tool's Trigger window—that is, a section of the Trigger window that provides trigger information for a specific system.

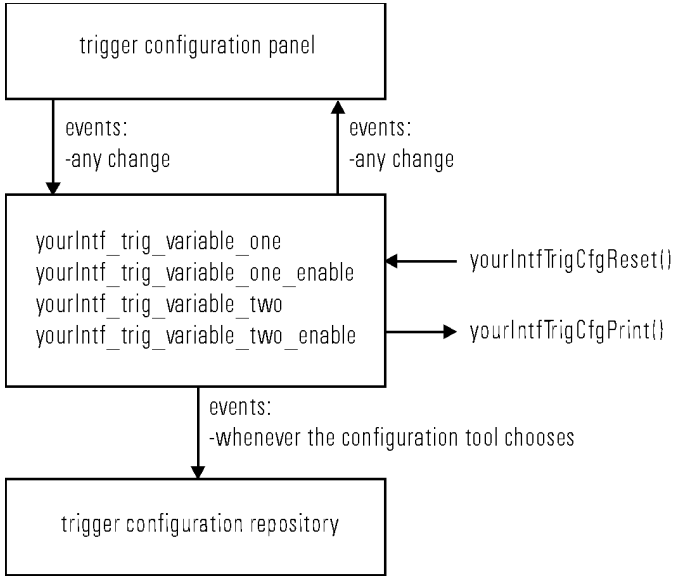
Trigger panels are optional. If the target system does not have any system-specific trigger information, you can disable trigger panel functionality for the system by setting the *trigger* flag in the **supports** method to 0 (see "yourIntfClass::supports" on page 7-43). With this flag set to 0, you do not need to define any trigger panel procedures for the target system, and you may skip the rest of this chapter.

If the target system needs system-specific trigger information, you must enable trigger panel functionality for the system by setting the *trigger* flag in the **supports** method to 1. When this flag is set to 1, you must supply the configuration tool with a collection of trigger panel procedures. When the configuration tool is running, it will call these trigger panel procedures as needed.

The configuration tool supports up to one trigger panel per type of target system. Trigger configuration data is maintained on a per-method basis.

Trigger Panel Data Flow

The following illustration shows how data flows between the trigger panel's Tk widgets, data variables that you define, and configuration repository. In this illustration, the target system type's short name is *yourIntf*.



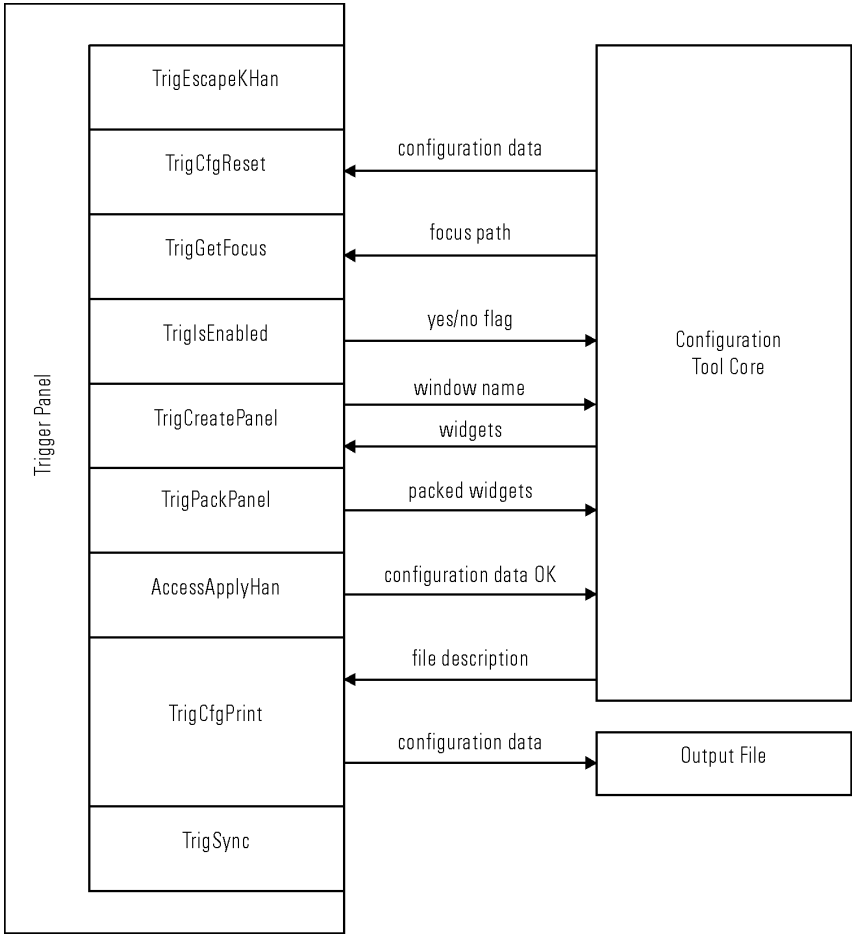
Trigger Panel Procedures

The following procedures are typically defined for trigger panels.

Procedure Name	Description	Page
yourIntfTrigCfgReset	Creates and initializes trigger panel variables.	6-10
yourIntfTrigCfgPrint	Prints trigger panel configuration data to a file.	6-11
yourIntfTrigGetFocus	Gets the current trigger focus.	6-13
yourIntfTrigCreatePanel	Creates the trigger panel.	6-14
yourIntfTrigPackPanel	Packs the trigger panel widgets together.	6-17
yourIntfTrigIsEnabled	Returns whether or not any trigger is enabled.	6-19
yourIntfTrigEscapeKHan	Handles user presses of the Escape key.	6-20
yourIntfTrigApplyHan	Handles user clicks of the Trigger window's Apply button.	6-22
yourIntfTrigSync	Synchronizes the state of the trigger panel's widgets.	6-24

Many of these procedures must be defined since they are called by the configuration tool. Others are called by the communication object and are needed to create a working trigger panel.

The following illustration shows the interaction between the trigger panel procedures, configuration tool's core, and output file.



To develop a system type-specific trigger panel, follow these steps:

Step 1: Identify and define the trigger sources and trigger filters required for the type of target system.

To do this, you'll need to classify the target system type's orientation. There are two basic orientations:

1. Systems that asynchronously send the mapped data in messages.

   These systems push the data. Enterprise Link typically has little or no control over which messages are received, when they are received, and in what order they are received. These systems are message-oriented.

   For such systems, the system type-specific trigger conditions usually specify some type of filtering criteria based on the content of the received messages—such as a message name, a message field name, or a message field data value. In theory, the receipt of the message is the trigger, but filtering criteria may choose to discard the message, pretending that none was received.

   Agilent Technologies' SAP Communication Object is message-oriented.

2. Systems that allow the mapped data to be pulled from the target system.

   These systems are database-oriented.

   For such systems, the system type-specific trigger conditions usually specify some type of event that causes Enterprise Link to get the data. The event itself is usually an asynchronous event message generated by the target system—such as a database value-changed event or an invalid-value alarm. Usually, the data server arranges for the target system to generate these asynchronous trigger messages by configuring the target system as appropriate.

   Agilent Technologies' RTAP Communication Object and Oracle Communication Object are database-oriented.

Step 2: Assign a variable name to each of the items identified in Step 1.

To avoid name collisions with variables defined by the configuration tool and its associated libraries, prefix each name with the target system type's short name. Also, to avoid name collisions with other variables associated with the type of target system, follow the short name prefix with the string *_trig*.

For example, if the target system type's short name is MYSYS and if two variables with flags are required, the variables should have the following names:

```
MYSYS_trig_variable_one
MYSYS_trig_variable_one_enable
MYSYS_trig_variable_two
MYSYS_trig_variable_two_enable
```

Step 3: Develop two Tcl procedures to initialize and print the data defined in Step 1.

For example, if the target system type's short name is MYSYS, the procedures should have the following names:

```
MYSYSTrigCfgReset
MYSYSTrigCfgPrint
```

For a description of how to develop these procedures, see "Developing the yourIntfTrigCfgReset Procedure" on page 6-10 and "Developing the yourIntfTrigCfgPrint Procedure" on page 6-11.

Step 4: Decide if you want to display the current trigger focus on the Add Mapping window. If you choose to do this, you'll have to develop a Tcl procedure that returns the current trigger focus when called. This procedure is optional.

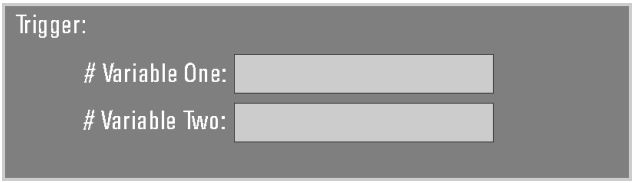
For example, if the target system type's short name is MYSYS, the procedure should have the following name:

```
MYSYSTrigGetFocus
```

For a description of how to develop this procedure, see "Developing the yourIntfTrigGetFocus Procedure" on page 6-13.

Step 5: Lay out the trigger panel.

The following shows a sample trigger panel built with Tk widgets.



Step 6: Develop Tcl procedures to create and pack the trigger panel.

For example, if the target system type's short name is MYSYS, the procedures should have the following names:

```
MYSYSTrigCreatePanel
MYSYSTrigPackPanel
```

For a description of how to develop these procedures, see "Developing the yourIntfTrigCreatePanel Procedure" on page 6-14 and "Developing the yourIntfTrigPackPanel Procedure" on page 6-17.

Step 7: Develop a Tcl procedure that allows the configuration tool to determine if any triggers for the target system are enabled. This procedure is called by the configuration tool to determine whether the trigger panel should be initially displayed expanded or unexpanded.

For example, if the target system type's short name is MYSYS, the procedure should have the following name:

```
MYSYSTrigIsEnabled
```

For a description of how to develop this procedure, see "Developing the yourIntfTrigIsEnabled Procedure" on page 6-19.

Step 8: Develop Tcl procedures to handle when the end user presses the Escape key or clicks on the Apply button in the trigger window, or when the global variable *el_app_obj_name* changes values.

For example, if the target system type's short name is MYSYS, the procedures should have the following names:

```
MYSYSTrigEscapeKHan
MYSYSTrigApplyHan
MYSYSTrigSync
```

For a description of how to develop the procedures, see "Developing the yourIntfTrigEscapeKHan Procedure" on page 6-20, "Developing the yourIntfTrigApplyHan Procedure" on page 6-22, and "Developing the yourIntfTrigSync Procedure" on page 6-24.

Developing the yourIntfTrigCfgReset Procedure

You must develop the **yourIntfTrigCfgReset** Tcl procedure. This procedure creates, then sets the trigger panel variables to their default state. These variables must be global for Tk's widgets to be able to access them.

The configuration tool calls this procedure during startup to create and initialize the global variables needed by the target system type's trigger panel.

The configuration tool accesses the global variable *yourIntf_trig_variables* to determine the names of the variables associated with the target system type's trigger panel. Therefore, it is very important for this variable to exist, be global, and be properly initialized.

The following example of a **yourIntfTrigCfgReset** procedure assumes that the Trigger window requires four variables: *yourIntf_trig_variable_one*, *yourIntf_trig_variable_one_enable*, *yourIntf_trig_variable_two*, and *yourIntf_trig_variable_two_enable*.

Example

```
proc yourIntfTrigCfgReset {} {
 global yourIntf_trig_variables

   # the variable names used by yourIntf trigger window
   # widgets
   set yourIntf_trig_variables [list\
       \${_sys}\${_domain}variable_one\
       \${_sys}\${_domain}variable_one_enable\
       \${_sys}\${_domain}variable_two\
       \${_sys}\${_domain}variable_two_enable]

   set _sys "yourIntf_"
   set _domain "trig_"
   set cmd "global [subst $yourIntf_trig_variables]"
   eval $cmd

   # initialize yourIntf variables
   set yourIntf_trig_variable_one <default value>
   set yourIntf_trig_variable_one_enable <default value>
   set yourIntf_trig_variable_two <default value>
   set yourIntf_trig_variable_two_enable <default value>
}
```

Developing the yourIntfTrigCfgPrint Procedure

You must develop the **yourIntfTrigCfgPrint** Tcl procedure. This procedure writes human-readable documentation about the current trigger configuration to a specified file descriptor. The written text should be easy to understand and complete.

The configuration tool calls this procedure whenever the end user chooses the Print menu item from the File menu in the configuration tool's main window or Edit Method window.

When the configuration tool calls this procedure, it passes to the procedure one parameter: an open file descriptor that all output should be written to. This procedure should only write to this file descriptor; it should not read from it nor close it.

To construct what is written, use the table formatting utility **utlTableRow** (see "utlTableHeader, utlTableRow, utlTablePut" on page 8-71). With this utility, the final output should be a table that looks something like the following:

Trigger Configuration		
System	Item	Value
	<name1> <name2>	<value1> <value2>
<yourIntf> <yourIntf>	<name3> <name4>	<value3> <value4>
	: :	: :

where

<name#> The name of Trigger window variables.

<value#> The current value of a Trigger window variable.

Empty lines, as shown in the example table above, may be inserted to separate related configuration parameters. This can improve the generated documentation's readability, especially for tables with many rows.

The following example of a **yourIntfTrigCfgPrint** procedure assumes that the Trigger window requires four variables: *yourIntf_trig_variable_one*, *yourIntf_trig_variable_one_enable*, *yourIntf_trig_variable_two*, and *yourIntf_trig_variable_two_enable*.

Example

```
proc yourIntfTrigCfgPrint f {
 global yourIntf_trig_variable_one \
        yourIntf_trig_variable_one_enable \
        yourIntf_trig_variable_two \
        yourIntf_trig_variable_two_enable

   # compose value strings
   set sys_str "yourIntf"

   if {$yourIntf_trig_variable_one_enable} \
             {set v1 $yourIntf_trig_variable_one
      } else {set v1 "<disabled>"}

   if {$yourIntf_trig_variable_two_enable} \
             {set v2 $yourIntf_trig_variable_two
      } else {set v2 "<disabled>"}

   # generate additional trigger documentation
   utlTableRow $f $sys_str "Var One" $v1
   utlTableRow $f $sys_str "Var Two" $v2
   utlTableRow $f
}
```

Developing the yourIntfTrigGetFocus Procedure

You can optionally develop the **yourIntfTrigGetFocus** Tcl procedure. This procedure allows the configuration tool to display the current trigger focus on the Edit Mapping window. This is very useful in target systems that asynchronously send data that is to be mapped in messages. Trigger focus directs the end user's attention to the relevant location on the Edit Mapping window. This reduces the possibility of confusion and error when the end user adds new mappings.

If you don't want to display the trigger focus for a specific type of system on the Edit Mapping window, you can disable trigger focus functionality for the target system type by setting the *trigger focus* flag in the **supports** method to 0 (see "yourIntfClass::supports" on page 7-43). With this flag set to 0, you do not need to define a trigger focus procedure, and you may skip this procedure.

If you want to display the trigger focus for a specific type of system, you must develop this procedure and enable trigger focus functionality for the target system type by setting the *trigger focus* flag in the **supports** method to 1.

With this flag set to 1, the configuration tool calls this procedure whenever it needs to update the Edit Mapping window. The configuration tool passes no parameters to this procedure, but expects this procedure to return a valid path specification in the form of a Tcl list.

The following example of a **yourIntfTrigGetFocus** procedure sets the trigger focus to the current value of *yourIntf_trig_variable_one*.

Example

```
proc yourIntfTrigGetFocus {} {
 global yourIntf_trig_variable_one \
        yourIntf_trig_variable_one_enable

   # if triggering is disabled
   if {!$yourIntf_trig_variable_one_enable} {return ""}

   return $yourIntf_trig_variable_one
}
```

Developing the yourIntfTrigCreatePanel Procedure

You must develop the **yourIntfTrigCreatePanel** Tcl procedure. This procedure creates the target system type's trigger panel. This panel is automatically inserted into the Trigger window as needed.

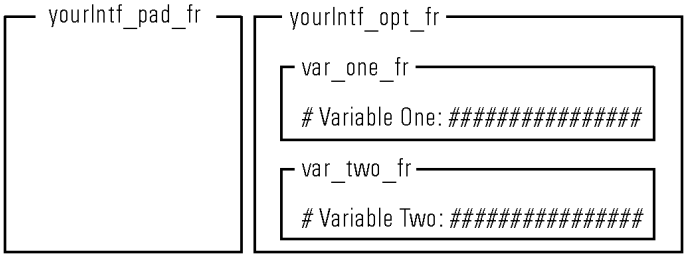
The configuration tool calls this procedure whenever the end user does one of the following:

- chooses the Trigger... menu item from the Edit Method window's Edit menu
- clicks on the Edit Trigger... button in the Edit Method window
- expands the target system type's trigger panel by enabling the Trigger window's yourIntf... check box

Before you can write this procedure, you must lay out the trigger panel's interface. Once you have done this, you must prepare your layout for use with Tk. The panel's labels will be built using Tk label widgets. The panel's text entry widgets will be built using Tk entry widgets. The panel's buttons will be built using Tk button widgets. All of these widgets must be grouped together using Tk frame widgets. Due to the way the Tk pack command works, each frame widget must enclose only rows or columns of Tk widgets.

The trigger panel's widgets should be pushed right 30 pixels. This can be done by creating an empty frame, and later placing it on the left-hand side of the panel in the **yourIntfTrigPackPanel** procedure (see "Developing the yourIntfTrigPackPanel Procedure" on page 6-17).

To make it easier to keep track of the position and nesting of trigger panel widgets, layout the Tk frames for the trigger panel on the layout you developed for the Access window (see "Developing the yourIntfAccessGui Procedure" on page 5-15). The following figure shows a sample layout for trigger panel frames.



When the configuration tool calls this procedure, it passes in one parameter: the name to use for the trigger panel's top-level window. Note that the configuration tool will never call **yourIntfTrigCreatePanel** if it already exists.

To process all of the window's widgets, call the **utlPrepareWidget** utility (see "utlPrepareWidget" on page 8-60). This utility ensures that the appearance and behavior of all widgets in the configuration tool are consistent. For entry and text widgets that are output only, use the utility's **-noEntry** option to ensure the widget's background color is appropriately set. When a check box or radio button widget controls whether or not an entry widget is enabled, use the utility's **-autoSelect** <*list*> option to ensure that the check box and radio button widgets automatically become enabled when the end user clicks a mouse button on the entry widget. The <*list*> option argument must be a Tcl list of all check box and radio button widgets that affect the state of the entry widget.

This procedure should return a Tcl list of text entry widgets that will be included in the Trigger window's focus traversal list. Focus traversal occurs when an end user first selects a text entry widget, then presses the **<Return>** key. Pressing the **<Return>** key causes the next entry widget to be selected for end user input.

This procedure must ensure that the trigger panel's widgets stay synchronized. If a check box is disabled or off, its corresponding entry widget should be disabled. Conversely, if a check box is enabled and on, its corresponding entry widget should be enabled. To do this, call the **yourIntfTrigSync** procedure (see "Developing the yourIntfTrigSync Procedure" on page 6-24) whenever the check box widget's state changes.

The following example of a **yourIntfTrigCreatePanel** procedure assumes that the Trigger window requires four variables: *yourIntf_trig_variable_one*, *yourIntf_trig_variable_one_enable*, *yourIntf_trig_variable_two*, and *yourIntf_trig_variable_two_enable*.

Example

```
proc yourIntfTrigCreatePanel p {
 global yourIntf_trig_variable_one \
        yourIntf_trig_variable_one_enable \         yourIntf_trig_
variable_two \
        yourIntf_trig_variable_two_enable

   set self $p
   if {![string match *. $p]} {set p "${p}."}

   # create frame widgets
   frame ${p}yourIntf_pad_fr
   frame ${p}yourIntf_opt_fr -relief groove
   frame ${p}var_one_fr
   frame ${p}var_two_fr
   utlPrepareWidget ${p}yourIntf_pad_fr ${p}yourIntf_opt_fr \
      ${p}var_one_fr ${p}var_two_fr

   # create sys one trigger type frame label
   label ${p}yourIntf_trig_lb -text "Trigger:" -padx 3 -anchor w
   utlPrepareWidget ${p}yourIntf_trig_lb

   # create "variable one" widgets
   checkbutton ${p}var_one_cb -text "Var One:" \
      -variable yourIntf_trig_variable_one_enable \
      -command [list yourIntfTrigSync $self]
   entry ${p}var_one_en -textvariable yourIntf_trig_variable_one
   utlPrepareWidget -autoSelect ${p}var_one_cb ${p}var_one_cb \
      ${p}var_one_en
   bind ${p}var_one_en <KeyPress-Escape> \
      [list yourIntfTrigEscapeKHan $self ${p}var_one_en]

   # create "variable two" widgets
   checkbutton ${p}var_two_cb -text "Var Two:" \
      -variable yourIntf_trig_variable_two_enable \
      -command [list yourIntfTrigSync $self]
   entry ${p}var_two_en -textvariable yourIntf_trig_variable_two
   utlPrepareWidget -autoSelect ${p}var_two_cb ${p}var_two_cb \
      ${p}var_two_en
   bind ${p}var_two_en <KeyPress-Escape> \
      [list yourIntfTrigEscapeKHan $self ${p}var_two_en]

   # return entry widget traversal chain
   return [list ${p}var_one_en ${p}var_two_en]
}
```

Developing the yourIntfTrigPackPanel Procedure

You must develop the **yourIntfTrigPackPanel** Tcl procedure. This procedure packs together the widgets created by the **yourIntfTrigCreatePanel** procedure.

The configuration tool calls this procedure whenever the end user does one of the following:

- chooses the Trigger... menu item from the Edit Method window's Edit menu
- clicks on the Edit Trigger... button in the Edit Method window
- expands the target system type's trigger panel by enabling the Trigger window's yourIntf... check button

Use the Tcl pack command to pack widgets. Be sure to use the appropriate **-expand** and **-fill** options to ensure that the widgets exhibit the correct behavior when the window is manually resized. Follow these guidelines:

- When packing frame widgets, supply pack the options **-expand 1** and **-fill both**.
- When packing label widgets, supply pack the option **-fill x**.
- When packing entry widgets, supply pack the options **-expand 1** and **-fill x**.
- When packing check box widgets, supply pack the option **-fill y**.

The following example of a **yourIntfTrigPackPanel** procedure assumes that the Trigger window requires four variables: *yourIntf_trig_variable_one*, *yourIntf_trig_variable_one_enable*, *yourIntf_trig_variable_two*, and *yourIntf_trig_variable_two_enable*.

Example

```
proc yourIntfTrigPackPanel p {

   set self $p
   if {![string match *. $p]} {set p "${p}."}

   # yourIntf frame
   pack ${p}yourIntf_pad_fr ${p}yourIntf_opt_fr -side left \
      -in $self -anchor n
   pack ${p}yourIntf_pad_fr -fill y -padx 15
   pack ${p}yourIntf_opt_fr -expand 1 -fill both

   # sys one trigger type groupings
   pack ${p}yourIntf_trig_lb ${p}var_one_fr \
      ${p}var_two_fr -side top -in ${p}yourIntf_opt_fr
   pack ${p}yourIntf_trig_lb -fill x
   pack ${p}var_one_fr ${p}var_two_fr -expand 1 -fill both -padx 5

   # trigger: variable one
   pack ${p}var_one_cb ${p}var_one_en -in ${p}var_one_fr \
      -side left
   pack ${p}var_one_en -expand 1 -fill x -pady 3

   # trigger: variable two
   pack ${p}var_two_cb ${p}var_two_en -in ${p}var_two_fr \
      -side left
   pack ${p}var_two_en -expand 1 -fill x -pady 3
}
```

Developing the yourIntfTrigIsEnabled Procedure

You must develop the **yourIntfTrigIsEnabled** Tcl procedure. This procedure indicates whether or not any triggers for the type of target system are enabled. If 0 is returned, the configuration tool assumes that all triggers are disabled. If 1 is returned, the configuration tool assumes that at least one trigger is enabled.

The configuration tool calls this procedure to determine whether the trigger panel should be initially displayed expanded or unexpanded. If any system type-specific triggers are enabled when the Trigger window is first displayed, the target system type's trigger panel is initially drawn expanded. Otherwise, the target system type's trigger panel is initially drawn unexpanded since it is not relevant for the current data mapping.

The following example of a **yourIntfTrigIsEnabled** procedure assumes that the Trigger window requires four variables: *yourIntf_trig_variable_one*, *yourIntf_trig_variable_one_enable*, *yourIntf_trig_variable_two*, and *yourIntf_trig_variable_two_enable*.

Example

```
proc yourIntfTrigIsEnabled {} {

 global yourIntf_trig_variable_one_enable \
        yourIntf_trig_variable_two_enable

   if {$yourIntf_trig_variable_one_enable ||
       $yourIntf_trig_variable_two_enable} {return 1
                                       } else {return 0}
}
```

Developing the yourIntfTrigEscapeKHan Procedure

You must develop the **yourIntfTrigEscapeKHan** Tcl procedure. This procedure handles when the end user presses the **<Escape>** key in a target system type's trigger panel entry widget. This procedure completes the entered value if unique, otherwise it displays a list of matching completions.

This procedure does not need to be called for entry widgets that do not support completion. But in the interests of consistency and usability, every entry widget should support completion if at all possible. Most of the configuration tool's entry widgets currently support completion. On the other hand, if none of the target system type's trigger panel entry widgets support completion, you do not need to define a trigger escape key handler procedure, therefore you may skip this procedure.

The configuration tool never calls this procedure. It is only called when the end user types the **<Escape>** key while the mouse cursor lies over an entry widget in the target system type's trigger panel. This causal relationship is set up in the **yourIntfTrigCreatePanel** procedure.

When Tcl calls the **yourIntfTrigEscapeKHan** procedure, it is usually passed two parameters: the name of the trigger panel and which entry widget to complete. This procedure should compose a list of completions for this widget, then pass this list to the **utlCompleteEntry** utility (see "utlCompleteEntry" on page 8-26).

If a list of appropriate completions could not be determined, such as when the target system's database is missing vital data or you cannot communicate with the target system, call the **utlCompleteEntryMsg** utility (see "utlCompleteEntryMsg" on page 8-27). This will temporarily display your informative error message in the entry widget that the end user is trying to complete.

If it is going to take a long time to compose the list of possible completions, use the **utlBusyCursor** and **utlIdleCursor** utilities (see "utlBusyCursor, utlIdleCursor" on page 8-17). These utilities allow you to temporarily display the end user's mouse cursor as an hourglass sprite.

Example

```
proc yourIntfTrigEscapeKHan {p w} {

   if {![string match *. $p]} {set p "${p}."}

   # fetch entered value
   set value [$w get]

   utlBusyCursor

   # compose a list of matching entries
          if {$w == "${p}val_one_en"} \
             {set items < compose Tcl list here >
   } else if {$w == "${p}val_two_en"} \
             {set items < compose Tcl list here >
   } else {
       utlIdleCursor
       return
   }

   utlCompleteEntry $p $w $items -title "Trigger Completions:"

   utlIdleCursor
}
```

Developing the yourIntfTrigApplyHan Procedure

You must develop the **yourIntfTrigApplyHan** Tcl procedure. This procedure verifies the values stored in the target system type's trigger configuration variables.

If any invalid values are found, this procedure should report the error and return. In this case, the displayed trigger configuration is not saved and the Trigger window remains mapped. Otherwise, this procedure should return without error, indicating that the target system type's portion of the trigger configuration is correct.

The configuration tool calls this procedure whenever the end user clicks on the Trigger window's Apply button.

When this procedure is called, it is passed one parameter: the name of the trigger panel to check.

When entry widgets have an associated enable/disable check box widget and the check box widget is disabled, do not check the current values of the dependent entry widgets. In this situation, illegal entry widget values are allowed.

Before saving the end user's changes, check for these common problems:

- empty entry widgets
- numeric entry widgets filled with non-numeric text
- non-numeric entry widgets filled only with numeric text
- invalid combinations of enabled checkbutton widgets

To detect and report the first and third problems, use the **utlChkAlpha** utility (see "utlChkAlpha" on page 8-20). To detect and report the first and second problems, use the **utlChkInt** utility (see "utlChkInt" on page 8-23). You'll have to detect the last problem yourself and use the Tcl error command to report the problem to the end user. Be aware that the configuration tool intercepts errors and displays themr in an Error window.

Once you've verified that the current values of the trigger variables are correct, press <**Return**>. The configuration tool saves the current values of the target system's trigger variables to a configuration file (assuming everything else entered on the Trigger window is correct).

The following example of a **yourIntfTrigApplyHan** procedure assumes that the Trigger window requires four variables: *yourIntf_variable_one*, *yourIntf_variable_one_enable*, *yourIntf_variable_two*, and *yourIntf_variable_two_enable*.

Example

```
proc yourIntfTrigApplyHan p {
 global yourIntf_variable_one \
        yourIntf_variable_one_enable \
        yourIntf_variable_two \
        yourIntf_variable_two_enable

   if {![string match *. $p]} {set p "${p}."}

   # check fields...
   if {$yourIntf_variable_one_enable} {
      utlChkAlpha $yourIntf_variable_one \
         -comment "Expected a symbolic database address." \
         [utlWidgetText ${p}var_one_cb]
   }

   if {$yourIntf_variable_two_enable} {
      utlChkAlpha $yourIntf_variable_two \
         -comment "Expected a symbolic database address." \
         [utlWidgetText ${p}var_two_cb]
   }
}
```

Developing the yourIntfTrigSync Procedure

You must develop the **yourIntfTrigSync** Tcl procedure. This procedure synchronizes the state of the target system type's trigger panel widgets.

This procedure is called whenever the state of a trigger panel widget changes. For example, if an entry widget has an associated enable/disable check box widget, the entry widget should be disabled when the check box widget is off and enabled when it is on. For nested widget dependencies, more complex widget enabling and disabling is required.

The configuration tool never calls this procedure. Rather, it is called when a trigger panel widget changes state. This causal relationship is set up in the **yourIntfTrigCreatePanel** procedure.

The following example of a **yourIntfTrigSync** procedure assumes that the Trigger window requires four variables: *yourIntf_trig_variable_one*, *yourIntf_trig_variable_one_enable*, *yourIntf_trig_variable_two*, and *yourIntf_trig_variable_two_enable*.

Example

```
proc yourIntfTrigSync p {
 global yourIntf_trig_variable_one_enable \
        yourIntf_trig_variable_two_enable

   if {![string match *. $p]} {set p "${p}."}

   if {$yourIntf_trig_variable_one_enable} \
      {${p}var_one_en configure -state normal
      } else {${p}var_one_en configure -state disabled}

   if {$yourIntf_trig_variable_two_enable} \
      {${p}var_two_en configure -state normal
      } else {${p}var_two_en configure -state disabled}
}
```

7

Class Reference

Class Reference

This chapter documents the communication classes for the Agilent Technologies Enterprise Link data server, configuration tool, and spooler.

The “elLinkClass” on page 7-25 documents the class for the data server’s core. The “elCommClass” on page 7-5 documents the data server’s base communication class and the modifications that you must make to the base class for your class.

The “yourIntfClass” on page 7-36 documents the class that you must write for the configuration tool.

The “elSpoolerClass” on page 7-34 documents the spooler’s base class for the elFIFOSpoolerClass and the elRASpoolerClass. The “elFIFOSpoolerClass” on page 7-21 documents the “first-in-first-out” (FIFO) spooler class. The “elRASpoolerClass” on page 7-30 documents the random-access (RA) spooler class.

Return Values

The return value for many of the methods in this chapter is “No return value is defined.” Without an explicit “return” statement, a Tcl function returns whatever value happened to be returned by the last command used in the function. This means that no particular return value is defined: the method does not consistently return anything.

Note

Your code should not rely on any values returned from a method with no defined return value, since the method may return some different value in a future release of Enterprise Link.

Current Working Directory

All of the methods of an Enterprise Link communication object that change the current working directory during their execution must be sure to restore the current working directory to its original value before returning. Failure to do this will result in undesirable behavior of the elserver process. For example, it will no longer be possible to stop the elserver process from the interactive configuration tool elconfig.

The current working directory must be restored whenever any communication object method returns successfully, unsuccessfully, and whenever a Tcl error occurs during method execution.The current working directory can be obtained by calling the Tcl procedure `pwd` and can be changed by calling the Tcl procedure `cd`.

Example

The Tcl method shown below demonstrates how to use the Tcl *catch* procedure to ensure that the current working directory is always restored to its original value:

```
method write args {

     # --- note the original working directory ---
     set orig_pwd [pwd]

     # --- do the write inside a "catch" command ---
     if {[catch {

               :
          cd <the desired directory>
               :
          <do write-related stuff here>

     # --- restore current working directory ---
     } rv]} {
          catch {cd $orig_pwd}
```

Return Values

```
        error $rv
    } else {
        cd $orig_pwd
    }

    return $rv
}
```

elCommClass

The **elCommClass** is the base class for all data server communication classes. When you write a data server communication class, you should inherit from this class. See Chapter 3, "Developing the Data Server Communication Object," for procedures and examples. This class provides useful default methods that will respond correctly for any methods you choose not to implement in your communication class. In addition, the base class provides two instance variables: **$discardInput** and **$discardOutput**.

elCommClass Methods

commit	Makes all **write** invocations since the last **commit** or **rollBack** permanent.
constructor	Initializes instance variables.
consumeOptions	Processes command-line arguments.
destructor	Shuts down communication with the target system.
getChildren	Returns the child node names as a Tcl list.
list2path	Converts a list to a path string.
open	Loads configuration information and prepares the object for use.
options	Queries and sets command-line option keywords.
path2list	Converts a path string to a list.
read	Reads values from the target system.
rollBack	Undoes all **write** invocations since the last **commit** or **rollBack.**
run	Completes trigger setup after all triggers have been passed to **setTrigger**.
selectionProcedures	Retrieves, deletes, executes or creates selection procedures.
setTrigger	Sets a trigger for the indicated method.
supports	Indicates whether or not a feature is supported.
usage	Returns command-line usage information.
write	Writes values to the target system.

elCommClass Data Members

$discardInput	Discard data received from the target system.
$discardOutput	Discard data that would be sent to the target system.

---

elCommClass::commit

commit

Return Value — No return value is defined.

Remarks — The base class's **commit** method just returns, which is the correct response when a communication class does not support a **commit** concept. If your target system does not support this concept, you do not need to define a **commit** method.

Your derived class's **commit** method should make the effects of all **write** invocations since the last **commit** or **rollBack** permanent in the target system. Your method should work hard to avoid raising a Tcl error. For example, if **elLinkClass::execute** has already invoked **commit** in several communication objects and then your communication object signals an error, very little error recovery can occur. **elLinkClass::execute** has no choice but to log an error and continue committing, because objects that have already been committed cannot be rolled back.

See Also — "elCommClass::write" on page 7-19, "elCommClass::rollBack" on page 7-14, and "elLinkClass::execute" on page 7-26.

---

elCommClass::constructor

constructor

Return Value — No return value is defined.

Remarks

The constructor method is invoked automatically when the object is first created and should carry out any simple initializations required to do **options**, **consumeOptions**, or **usage** argument parsing.

Most communication objects will not need a **constructor** method since all non-trivial initialization is done in the **open** method and trivial initialization of instance variables can be done directly in the instance-variable declaration in [incr Tcl].

---

elCommClass::consumeOptions

consumeOptions *varName*

Return Value

No return value is defined.

Parameters

varName The name of the variable containing the list of command-line arguments.

Remarks

The base class's **consumeOptions** method just returns, which is the correct response when a communication object doesn't use command-line arguments. Only override this method with your own when you have command-line options to process.

Your derived class's **consumeOptions** method should search a list of command-line arguments, removing from the list any that the communication object recognizes. If an error in command-line argument parsing is detected (for example, missing arguments or an invalid combination of arguments), your method should raise a Tcl error condition.

You can use the **utlGetArg** utility to remove command-line option keywords from *$varName*. You can use the **utlPeekArg** utility to parse keywords and leave them in *$varName*. If you use these utilities, you must do an

```
upvar $varName args
```

since the utilities assume they are working on a local variable *args*. Do not call the **utlArgEnd** utility since other communication objects may not yet have had their **consumeOptions** method invoked at the time yours is invoked.

See Also

"elCommClass::options" on page 7-11, "elCommClass::usage" on page 7-18, and "utlGetArg, utlPeekArg, utlArgEnd" on page 8-43.

elCommClass::destructor

destructor

Return Value — No return value is defined.

Remarks — The **destructor** method is invoked automatically when the object is destroyed. It should shut down all communication links with the target system. Currently, communications are only destroyed when the data server exits.

elCommClass::discardInput

$discardInput

Remarks — When this variable is 1, the communication object should discard data received from the target system. By default, this variable is 0.

When this variable is 1, the **setTrigger** method should ignore all invocations. When both the **$discardInput** and the **$discardOuput** variables are 1, you should not even try to communicate with the target system.

elCommClass::discardOutput

$discardOutput

Remarks — When this variable is 1, the communication object should discard data that would otherwise have been sent to the target system. By default, this variable is 0.

When both the **$discardInput** and the **$discardOuput** variables are 1, you should not even try to communicate with the target system.

elCommClass::getChildren

getChildren [**-filter** *pattern*] *node*

Return Value — Must return as a Tcl list the names of the child nodes that are actually available from the source system.

Parameters — **-filter** *pattern* Only return names of children that match *pattern*.

node Full path name to the desired parent node, expressed as a Tcl list.

Remarks — So that the data server can use communication object-independent (also known as *selection*) procedures, all communication objects supporting dynamic mapping must also support the **getChildren** method. Your **getChildren** method must get the names of the children under *$node*. The returned list of child names must not include any path information.

By definition, the root node of the name space tree is the empty string. Therefore, [$commObject getChildren ""] returns the names of all nodes lying directly under the root node.

Generic selection procedures invoke this method whenever they need to acquire the names of child nodes.

elCommClass::list2path

list2path *pathAsList*

Return Value — Returns the human-readable version of *$pathAsList*.

Parameters — *pathAsList* A list specifying a path in the logical name space.

Remarks — The base class's **list2path** method converts *pathAsList* into a human-readable file system path by inserting solidus characters (/) between each item in the list. For example, `{a b c}` is turned into `/a/b/c`.

This method is invoked by **elLilnkClass::execute** and other methods when paths need to be made part of error messages.

Your derived class's **list2path** method should convert *pathAsList* into something that makes sense for your end users.

See Also "elCommClass::path2list" on page 7-11

---

elCommClass::open

open

Return Value No return value is defined.

Remarks The base class's **open** method loads the communication object's *obj*_dataflow.cfg configuration file and sets the **$discardInput** and **$discardOutput** instance variables accordingly. This is the correct response if your derived class does not have an **open** method.

If your derived class does have an **open** method, the *obj*_dataflow.cfg file must still be read. The most reliable way to read the file is to invoke this base class's **open** method from your derived class's **open** method. For example,

```
itcl_class myDerivedClass {
   inherit elCommClass
   method open {} {
      elCommClass::open
      ...
```

If the base class's **open** method sets both **$discardInput** and **$discardOutput** to 1, then your **open** method should not even attempt to open a connection to the target system. This is useful when debugging other communication objects.

If the configuration tool will be developed to support the viewing and editing of system-specific access information, the data server's **open** method needs to read and load the access configuration file before opening the connection to the target system. If this configuration file does not exist, you'll have to either report an error or use default access configuration parameters. If you use default values, they should match the configuration tool's Access window defaults. If **open** raises a Tcl error condition, the data server prints an error message and exits.

See Also "elCommClass::destructor" on page 7-8.

elCommClass::options

options *optkeys*

Return Value — Returns the current command-line option keyword names as a Tcl list.

Parameters — *optkeys* New names for command-line option keywords as a Tcl list. The list must be exactly as long as the number of command-line keywords the object supports. Each option in the list will replace the option that occupies the same position in the current command-line keyword list.

Remarks — The base class's **options** method just returns, which is the correct response when a communication object doesn't use any command-line arguments. Only override this method with your own when you have command-line options to process.

Your derived class's **options** method should get or set the command-line option keywords supported by this object. The configuration tool invokes this method to determine what command-line option keywords are currently supported and also to assign new command-line option keywords when there are keyword collisions with either the configuration tool's command-line option keywords or some other communication object's command-line option keywords.

See Also — "elCommClass::consumeOptions" on page 7-7, "elCommClass::usage" on page 7-18, and the variable *argv* in your Tcl documentation.

elCommClass::path2list

path2list *path*

Return Value — Returns *$path* as a Tcl list.

Parameters — *path* The path to convert to a Tcl list.

Remarks — The base class's **path2list** method converts *$path* to a Tcl list. This method implements exactly the opposite conversions that are implemented in the **list2path** method.

The data server invokes this method to convert paths expressed with a syntax more easily understood by the user to paths expressed as Tcl lists.

See Also "elCommClass::list2path" on page 7-9

---

elCommClass::read

read {*configuredMethodName1 ...*} *srcArrayName*

Return Value No return value is defined.

Parameters {*configuredMethodName1 ...*} The list of configured method names to acquire data for and store in the source array.

srcArrayName The name of the source array to store the target system data in.

Remarks The base class's **read** method just returns, which is the correct response when a target system only accepts data and never sends any data back to your communication class.

Your derived class's **read** method should read values from the target system and store them in the source array. The **elLinkClass::execute** method invokes your **read** method when it needs data values from your object to execute the list of configured methods. Your **read** method must use **elLinkClass::methodInfo** to retrieve the list of name-space paths that specify which data values are needed and must retrieve those data values from the target system. Your method must also determine which, if any, selection procedures need to be invoked, then invoke those selection procedures and note the paths they return. Your **read** method must then retrieve any needed data values from the target system.

In database-oriented communication objects, your **read** method must query the database for the indicated source paths. In message-oriented communication objects, your **read** method must find source values for all the indicated source paths in the most recent message received from the target system.

The source array is a nested array. Tcl does not support a simple src(a)(b)(c) syntax for nested arrays, but nested arrays are a supported part of the Tcl language (see the `upvar` command). If you do not want to manipulate nested arrays directly, you may use **utlGetArray** and **utlSetArray** as illustrated in the vector and filters examples.

Conceptually, the structure of the source array is

```
src(srcPath) = value
src(srcPath)(rowNum) = value
src(srcPath)(size) = number of rows
src(srcPath)(methodName)(rowNum) = value
src(srcPath)(methodName)(size) = number of rows
```

Scalar values are stored with only one level of nesting. Therefore, you can manipulate them using the standard Tcl array reference syntax.

```
upvar $srcArrayName src
set src($path) $value
```

Vectors and columns of tables are represented as Tcl arrays nested within the source array. Therefore, to manipulate these arrays you need utilities like **utlGetArray** and **utlSetArray**. These vector and column arrays always start their row numbers at zero and their size element's value must be one greater than the highest-numbered element. For example,

```
upvar $srcArrayName src
set i 0
foreach value [allValuesInColumn $path] {
   utlSetArray src $path $i $value
      incr i
}
utlSetArray src $path size $i
```

Notice that the size element is not the number of elements in the vector or column, rather it is one greater than the largest array index. In the example above, these quantities are identical only because there are no missing entries in the column being stored in the source array.

If you have implemented any filters in your trigger conditions, you need a third level of array nesting to indicate which vectors or columns belong to the filters associated with each configured method. For example,

```
foreach methodName $configuredMethodNames {
   set i 0
   foreach value [allValuesInColumn $path $methodName] {
      setArray src $path $methodName $i $value
         incr i
   }
   setArray src $path $methodName size $i
}
```

The **read** method may refresh the values of paths in the source array that already have values.

The Tcl *${srcArrayName}_select* variable is an array indexed by a string made up of a selection procedure's name and arguments. Each element of this array contains the list of source addresses the selection procedure found when it was executed. Each of the source addresses are in the form of a Tcl list. For example:

```
upvar ${array_name}_select select

# --- for each selection procedure called by $method ---
foreach info [elLink methodInfo select $method] {
   set proc_and_args [lindex $info 1]

   # extract selection procedure's name and arguments
   set proc [lindex $proc_and_args 0]
   set args [lindex $proc_and_args 1]

   # invoke selection procedure and store the results
   set select($proc_and_args) [<invoke selection procedure $proc here>]
}
```

Errors

In message-oriented communication objects, the **read** method is responsible for determining whether all the values that arrived in a message were either mapped somewhere or were explicitly discarded. The **read** method should log an error message for every unused value, providing the logical name space path and value.

As a rule, the **read** method should not log any other error messages. The **elLinkClass::execute** method diagnoses missing values in the source array and two error messages should not be logged for each missing value. Furthermore, **execute** can use context like the user-configured error-handling scheme for the method to respond correctly when an error is detected. The **read** method has a harder time responding correctly. Therefore, the **read** method should raise an error condition or log an error only in those rare cases when something happens that **execute** is not likely to diagnose effectively.

See Also

"elCommClass::write" on page 7-19, "elLinkClass::execute" on page 7-26, "utlGetArray" on page 8-46, and "utlSetArray" on page 8-63.

elCommClass::rollBack

rollBack

Return Value

No return value is defined.

Remarks

The base class's **rollBack** method just returns, which is the correct response when a communication class does not support a roll back concept. If your communication class does not support the roll back concept, you do not need to define this method.

Your derived class's **rollBack** method should undo the effects of all the **write** invocations since the last **commit** or **rollBack** invocation.

See Also

"elCommClass::commit" on page 7-6, "elCommClass::read" on page 7-12, and "elLinkClass::execute" on page 7-26.

---

elCommClass::run

run

Return Value

No return value is defined.

Remarks

The **run** method is invoked by the data server once all triggers have been passed to the **setTrigger** method. Pairing the **run** method with the **setTrigger** method ensures that the communication object proceeds with any required actions only after it has accepted all trigger information.

Using the **setTrigger** and **run** method pair is useful for configurations in which setting triggers one by one is either inefficient or impossible, as is the case with database-oriented messages.

When the **$discardInput** variable is set to 1, all input to this communication object is being discarded and this method or method pair should ignore all invocations.

See Also

"elCommClass::setTrigger" on page 7-17.

elCommClass::selectionProcedures

selectionProcedures

selectionProcedures *procedureName*

selectionProcedures *procedureName arguments*

selectionProcedures *procedureName arguments body apiVersion*

Return Value

The **selectionProcedures** method without parameters must return the names of all available selection procedures as a Tcl list of zero or more three-item lists:

```
{<procedureName> <arguments> <apiVersion>} {<procedureName> <arguments>
<apiVersion>} ...
```

The Tcl list should include selection procedures originating in both the communication object and the data server.

The **selectionProcedures** method with only the *procedureName* parameter must return a Tcl list of the names of deleted procedures:

```
<procedureName> <procedureName> ...
```

The **selectionProcedures** method with the *procedureName* and *arguments* parameters must return the selection procedures invoked, error messages encountered, and the selected sources as a Tcl list of zero or more three-item lists:

```
{{<procedureName> <arguments>} {errCode} {<selectedSources>}}
{{<procedureName> <arguments>} {errCode} {<selectedSources>}} ...
```

The **selectionProcedures** method with the *procedureName*, *arguments*, *body* and *apiVersion* parameters returns a Tcl list of the newly created selection procedures:

```
<procedureName> <procedureName> ...
```

Parameters

procedureName The name of the selection procedure.

arguments The arguments used by the selection procedure.

body The body text defining the selection procedure.

apiVersion The application programming interface (API) version of the arguments and the values they return. In the E.02.20 version of Enterprise Link, the API version is 1.

Remarks

The **selectionProcedures** method is invoked by the data server in order to query, create, execute, and delete selection procedures.

If only procedure names are supplied as a parameter, this method deletes selection procedures. If procedure names and arguments are supplied, this method executes selection procedures. If procedure names, procedure arguments, procedure bodies, and API versions are supplied, this method creates selection procedures.

---

elCommClass::setTrigger

setTrigger *configuredMethodName*

Return Value

No return value is defined.

Parameters

configuredMethodName The name of the configured method to associate with the trigger condition.

Remarks

The base class's **setTrigger** method just returns, which is the correct response when a communication class does not provide any triggers and does not support time-based triggers.

The **setTrigger** method can behave in one of two ways. In the first case, the **setTrigger** method completes a series of actions every time it receives trigger information. In the second case, the **setTrigger** method collects trigger information until the **run** method is invoked by **elCommClass::execute**.

In the first case, your derived class's **setTrigger** method should set a trigger for the indicated method. The **setTrigger** method associates a configured method name with a trigger condition. If necessary, the **setTrigger** method tells the target system to notify the communication object of the occurrence of the trigger condition and it establishes a Tcl event handler to receive such notification.

In the second case, the setTrigger method does all the activities described above, except the **run** method—not the setTrigger method—tells the target system to notify the communication object of the occurrence of the trigger condition and the **run** method then establishes a Tcl event handler to receive the notification.

The invoker's local variables contain the description of the trigger condition. Your **setTrigger** method will have to use the Tcl `upvar` command to extract the values of these variables. The names and meanings of these variables are part of the interface you defined between the configuration tool and the data server. See "Configuration Tool/Data Server Interface" on page 1-19.

Each communication object capable of initiating a configured method execution must override this method or method pair.

When the **$discardInput** variable is set to 1, all input to this communication object is being discarded and this method or method pair should ignore all invocations.

See Also "elCommClass::run" on page 7-15

---

elCommClass::supports

supports *selection procedures*

Return Value If a parameter is not supplied, the supports method must return a list of all parameters that are potentially supported. If a parameter is supplied, the supports method must return 1 if the parameter is supported and 0 if it is not.

Parameters *selection procedures* Selection procedures. Setting this to 0 dims the Select checkbox in the Edit Mapping window.

Remarks Your **supports** method should return 0 if the functionality specified by each parameter is unrecognized. The data server invokes this method to determine the facilities provided by this communication object.

---

elCommClass::usage

usage

Return Value Returns the list of command options.

Remarks

The base class's **usage** method returns {}, which is the correct response when a communication object doesn't use any command-line arguments. Only override this method with your own when you have command-line options to process.

Your derived class's **usage** method should return a list containing command-line argument usage information. The list is actually a list of lists with the following format:

```
{{arg1summary arg1details} ...}
```

Each argument summary is a list like the following:

```
{[-e envName]}
```

The brackets ([]) indicate an optional argument. Each details is a list explaining what the argument means. New-line characters and any subsequent space or tab characters in details are translated into space characters. When displayed to the end user, the details string will have its lines wrapped appropriately.

---

elCommClass::write

write *valuesVarName*

Return Value

No return value is defined.

Parameters

valuesVarName The name of a variable in the invoking procedure whose value is a list like the following:

```
{{toPath1 value1} ... {toPathN valueN}}
```

toPath: where to store the value
value: the value to store

Remarks

The base class's **write** method raises a Tcl error condition whenever anything is written to it, which is the correct response for a read-only communication class.

Your derived class's **write** method should write values to the target system. In communication classes with no concept of a **commit** method, the values are written to the target system immediately. In classes with a **commit** method, the values are only written to the target system after the **commit** method is invoked or the values are undone when the **rollBack** method is invoked.

The write method is invoked once for each configured method that executes. When values are written to a table, your **write** method should interpret all writes to the same table as being to the same row of that table. The next **write** invocation starts another row. In message-oriented communication objects, when values are written to a message, your **write** method should interpret all writes to the same message as being to the same instance of that message. The next **write** invocation starts another message. For messages that contain tables, the next **write** invocation generally starts a new row in the table and the message is not sent to the target system until **commit** is invoked.

If an error is encountered within the **write** method, you can discover how to deal with the errors using the following:

```
set how [elLink methodInfo error [elLink curMethod]]
```

A how value of

- continue means log the error and write the other values in the list to the target system.
- abandonMethod means log the error and return, without writing anything to the target system.
- abandonAllMethods means do not log the error, but raise a Tcl error condition with an informative error message. Your invoker is always **elLinkClass::execute**. It will log the error, abandon work on all other methods, and invoke **rollBack** for every object that was written to.

See Also

"elCommClass::read" on page 7-12, "elCommClass::commit" on page 7-6, "elCommClass::rollBack" on page 7-14, "elLinkClass::execute" on page 7-26, "elLinkClass::methodInfo" on page 7-28, and "elLinkClass::curMethod" on page 7-25.

elFIFOSpoolerClass

The **elFIFOSpoolerClass** inherits from the **elSpoolerClass**. The **elSpoolerClass** loads configuration files created by the configuration tool and provides utilities to query that configuration (see "elSpoolerClass" on page 7-34). In addition to the methods provided by the **elSpoolerClass**, the **elFIFOSpoolerClass** implements a conventional "first-in-first-out" (FIFO) spooler. If message limits are specified, older messages are discarded to make room for newer messages.

The spooler lets you read a number of messages without consuming them by creating a cursor: a pointer into the spool file. Later, you can make any cursor "the" cursor by invoking the **commit** method.

Caution

Cursors are necessarily very short-lived. They may safely be passed only to a number of consecutive **read** invocations and then to a **commit** invocation. All cursors become invalid if you read from the spool file without a cursor, append to the spool file in between uses of a particular cursor, or do anything else that may move data within the spool file.

elFIFOSpoolerClass Methods

constructor	Creates an instance of the FIFO spooler and initializes it.
byteCount	Returns the number of in-use bytes in the spool file.
commit	Sets the spool file cursor to the value of the cursor variable name.
msgCount	Returns the number of messages queued in the spooler.
print	Prints a representation of the spool file object and of the spool file contents.
read	Reads the oldest message in the spooler.
write	Writes a message to the end of the spool file.

elFIFOSpoolerClass::constructor

elFIFOSpoolerClass *spooler configFileName spoolFileName*
[**-errorHandling** *error|warning*]

Return Value — No return value is defined.

Parameters — *spooler* The name of the spooler object to create. The remaining parameters are the same as the ones for "elSpoolerClass::constructor" on page 7-34.

Remarks — Creates an instance of the FIFO spooler and initializes it.

elFIFOSpoolerClass::byteCount

byteCount

Return Value — Returns the number of in-use bytes in the spool file.

elFIFOSpoolerClass::commit

commit *cursorVarName*

Return Value — No return value is defined.

Parameters — *cursorVarName* The name of the variable containing the cursor to commit.

Remarks — Sets the spool file cursor to the value of the cursor variable name in the invoker's stack frame. The cursor must refer to a legitimate message position in the spool file. If it does not, the spool file will be corrupted.

elFIFOSpoolerClass::msgCount

msgCount *cursorVarName*

Return Value — Returns the number of messages queued in the spooler. If *cursorVarName* is specified, returns the number of messages in the spooler following the cursor.

Parameters — *cursorVarName* The name of a variable containing the cursor to count from.

elFIFOSpoolerClass::print

print *fd*

Return Value — No return value is defined.

Parameters — *fd* The name of the file descriptor to print to. If *fd* is not specified, the file is printed to standard out.

Remarks — Prints a representation of the spool file object and of the spool file contents for debugging.

elFIFOSpoolerClass::read

read [**-timeStamp** *timeStampVarName*] [**-cursor** *cursorVarName*]

Return Value — Returns the oldest message in the spooler.

Parameters — **-timeStamp** *timeStampVarName* The name of the variable to assign to the time stamp value associated with the spooled message being read. Every message in the spooler has a timestamp containing the time the message was spooled.

-cursor *cursorVarName* The name of a variable containing the cursor to read from. If the variable is unset or has no value, a cursor is created that refers to the message just after the message being read. If the variable is set, the message read is the one the cursor refers to and the cursor is advanced to the next message.

Remarks

Reads the oldest message in the spooler. When no **-cursor** argument is specified, the message read is removed from the spool file.

---

elFIFOSpoolerClass::write

write *message*

Return Value

No return value is defined.

Parameters

message The message to store in the spool file.

Remarks

Writes a message to the end of the spool file. If the write causes an error, the error will be handled as specified by the **elSpoolerClass::errorHandling** method.

See Also

"elSpoolerClass::errorHandling" on page 7-35.

elLinkClass

The only instance of the **elLinkClass** is the elLink object. The elLink object is the core of the data server. It executes configured methods and provides utilities for communication objects.

elLinkClass Methods

configDir	Returns the path of the configuration directory.
curMethod	Returns the name of the method currently being executed.
execute	Executes a list of configured methods.
log	Saves a message to the appropriate log.
methodInfo	Returns information about a specific configured method.
version	Returns the version of software running in the data server.

elLinkClass::configDir

configDir

Return Value

Returns the path of the directory that the data server is currently accessing configuration files from.

elLinkClass::curMethod

curMethod

Return Value

Returns the name of the method currently being executed by the **execute** method. Returns {} if the **execute** method is not active.

elLinkClass::execute

execute *methodList*

Return Value — Returns 0 if no error or a recoverable error occurred, and 1 if an error that was not recoverable occurred. A recoverable error means not all methods were abandoned. A nonrecoverable error means all methods were abandoned and **rollBack** was called.

Parameters — *methodList* The list of configured method names to execute.

Remarks — Executes a list of configured methods. When a communication object detects that a trigger condition has been satisfied, it invokes **execute** and passes it the list of all configured method names whose trigger condition was satisfied. For any given stimulus from the target system, **execute** should be invoked exactly once. For example, if a value changed in the target system or a message was received from the target system, all the configured method names whose trigger conditions were satisfied by that stimulus should be passed to **execute**.

See also — *abandonAllMethods* in "elLinkClass::methodInfo" on page 7-28.

elLinkClass::log

log *messageType who msgToEval*

Return Value — Returns the error message that was printed.

Parameters — *messageType* The type of message being logged:

- *in* A trace of data that was just transmitted from the target system to a communication object. If trace logging is enabled, the trace will be written to the trace log. If trace logging is not enabled, the trace will be ignored.
- *out* A trace of data that was just transmitted to the target system from a communication object. If trace logging is enabled, the trace will be written to the trace log. If trace logging is not enabled, the trace will be ignored.

- *to* A trace of data that was just passed as an argument to a communication object's **write** method. Communication objects never use *to:* only the **execute** method uses *to*. If trace logging is enabled, the trace will be written to the trace log. If trace logging is not enabled, the trace will be ignored.
- *error* A message describing a serious error condition that may lead to a loss of data in the data server or to some other serious malfunction. The message will be written to the error log and if the user has configured it, to the trace log as well.
- *warning* A message describing a malfunction that has been temporarily corrected or worked around but may lead to incorrect operation some time in the future. The message will be written to the error log and if the user has configured it, to the trace log as well.
- *verbose* A message useful for debugging purposes that describes the progress of the data server or a communication object. These messages are written to the error log if the data server was started with the **-verbose** command line argument. The message will also be written to the trace log if the user configured the trace log to include the error log.

who Indicates which subsystem or object is generating the message. In communication objects, it is almost always *$this*: the name of the communication object.

msgToEval The message to evaluate and then print.

Remarks

Saves a message to the appropriate log. The **log** method accepts an unevaluated string as an argument, therefore complex messages can be buried inside an unevaluated Tcl list. If the **log** method determines that the string should be logged, it first passes the string to the Tcl subst command. In other words, the code only evaluates the list with the Tcl subst command if the message is really needed. For example,

```
# Trace the message received from the target system
elLink log in {[
   set message {}
   foreach field $fieldsFromApplication {
      append message "$field:\t$dataFromApplication($field)\n"
   }
   set message]}
```

By evaluating the message only when it is about to be printed, the **log** method reduces data server processing costs by not spending time creating long tracing messages only to discard them because tracing is disabled.

elLinkClass::methodInfo

methodInfo *modifier methodName*

Return Value — Returns the information specified by *modifier* and *methodName*.

Parameters — *modifier* The type of information to retrieve from the configured method:

- *variable* Returns the variable mapping information as a list of pairs:

  ```
  {{srcPath1 dstPath1} {srcPath2 dstPath2} ...}
  ```

- *constant* Returns the constant mapping information as a list of pairs:

  ```
  {{dstPath1 constant1} {dstPath2 constant2} ...}
  ```

- *select* Returns dynamic mapping information as a list of pairs:

  ```
  {{dstInfo1 srcInfo1} {dstInfo2 srcInfo2} ...}
  ```

  In this list, `srcInfo1` is itself in the form of a list specifying the procedure name and its arguments:

  ```
  {procName {arg1 arg2 ...}}
  ```

- *discard* Returns the discard information as a list:

  ```
  {path1 path2 ...}
  ```

- *from* Returns the name of the communication object providing the data.
- *to* Returns the name of the communication object that the data is being sent to.
- *fileName* Returns the name of the configuration file containing the configured method.
- *name* Returns the human-readable version of the configured method name.
- *error* Returns the configured error-handling regimen for mapping and other errors encountered during execution of the configured method. *error* returns one of the following:
  - *continue* Log the error and try to continue executing the configured method.
  - *abandonMethod* Log the error and return without writing anything to the target system.

- *abandonAllMethods* Do not log the error, but raise a Tcl error condition with an informative error message. Your invoker is always **elLinkClass::execute**. It will log the error, abandon work on all other methods, and invoke **rollBack** for every object that was written to.

methodName The name of the method to retrieve information from.

Remarks Retrieves specific information about a configured method.

---

elLinkClass::version

version

Return Value Returns the version of software running in the data server.

elRASpoolerClass

The **elRASpoolerClass** inherits from the **elSpoolerClass**. The **elSpoolerClass** loads configuration files created by the configuration tool and provides utilities to query that configuration (see "elSpoolerClass" on page 7-34). In addition to the methods provided by the **elSpoolerClass**, the **elRASpoolerClass** implements a random-access (RA) spooler. The RA spooler is slower than the FIFO spooler. If message limits are specified, older messages will be discarded to make room for newer messages.

Caution

File system block sizes cannot be reliably determined over NFS between different vendor's computers, and the RA spooler uses the file system block size to calculate how much space each message consumes. If your spool file is accessed through NFS, the RA spooler size calculations may malfunction and your spool file may consume either much more or much less space than you configured.

Message identification names must not contain ',' '/' or '.' characters.

elRASpoolerClass Methods

constructor	Creates an instance of the RA spooler and initializes it.
match	Returns the identification names of all the spooled messages that match the starname-expression *globExpr*.
print	Prints a representation of the spool object and of the spooled messages for debugging.
read	Reads the spooled message identified by *id*.
remove	Removes the message identified by *id* from the spooler.
write	Writes the message identified by *id* to the spooler.

elRASpoolerClass::constructor

elRASpoolerClass *spooler configFileName spoolFileName*
[-**errorHandling** *error|warning*]

Return Value — No return value is defined.

Parameters — *spooler* The name of the spooler object to create. The remaining parameters are the same as the ones for "elSpoolerClass::constructor" on page 7-34.

Remarks — Creates an instance of the RA spooler and initializes it.

elRASpoolerClass::match

match *globExpr*

Return Value — Returns the identification names of all the spooled messages that match the star-name expression *globExpr*.

Parameters — *globExpr* The name of the star-name expression to match spooled messages with.

Remarks — For additional information, refer to the string match command in your Tcl documentation.

elRASpoolerClass::print

print *fd*

Return Value — No return value is defined.

Parameters — *fd* The name of the file descriptor to print to. If *fd* is not specified, the file is printed to stdout.

Remarks — Prints a representation of the spool object and of the spooled messages for debugging.

elRASpoolerClass::read

read *id*

Return Value — Returns the spooled message identified by *id*. If the message identified by *id* does not exist, returns an error.

Parameters — *id* The identification name of the spooled message to be read.

Remarks — Reads the spooled message identified by *id*. The message read is not removed from the spool.

elRASpoolerClass::remove

remove *id*

Return Value — No return value is defined.

Parameters — *id* The identification name of the message to be removed from the spooler.

Remarks — Removes the message identified by *id* from the spooler. Returns successfully, even if the message identified by *id* does not exist.

elRASpoolerClass::write

write *message id*

Return Value — No return value is defined.

Parameters — *message* The message to be written to the spooler.

id The identification name of *message*.

Remarks — Writes the *message* identified by *id* to the spooler. If the write causes an error, the error will be handled as specified by the **elSpoolerClass::error Handling** method.

See Also — "elSpoolerClass::errorHandling" on page 7-35.

elSpoolerClass

This is the spooler base class for the **elFIFOSpoolerClass** and the **elRASpoolerClass**. This class loads configuration files created by the configuration tool and provides utilities to query that configuration.

elSpoolerClass Methods

constructor	Creates an instance of the spooler and initializes it.
errorHandling	Sets the error handling regime for the spooler.
isEnabled	Returns 1 if the spooler is enabled, 0 otherwise.
maxByteCount	Returns the maximum number of bytes the spool file is configured to contain.

elSpoolerClass::constructor

elSpoolerClass::constructor *configFileName spoolFileName* [**-errorHandling** *error* | *warning*]

Return Value — No return value is defined.

Parameters

configFileName The name of the file the configuration tool created that contains the configuration for this spooler object.

spoolFileName The name of the file to contain the spooled data.

-errorHandling Specifies an initial value for the **elSpoolerClass::errorHandling** method.

Remarks — **elSpoolerClass::spooler** is the syntax for invoking the base class constructor from within a constructor in a derived class. The base class constructor loads the configuration file and performs general initialization.

See Also — "elSpoolerClass::errorHandling" on page 7-35.

elSpoolerClass::errorHandling

errorHandling *error warning*

Return Value — If a parameter is not specified, the current parameter is returned.

Parameters — *error* Raises a Tcl error condition whenever a message is about to be discarded and does not discard the message. This means that if the spooler is full, you cannot add messages to the spooler until you remove some old ones.

warning Logs a warning message to the error log whenever a message is about to be discarded, then discards the message.

Remarks — Sets the error handling regime for the spooler.

elSpoolerClass::isEnabled

isEnabled

Return Value — Returns 1 if the spooler is enabled, 0 otherwise.

elSpoolerClass::maxByteCount

maxByteCount

Return Value — Returns the maximum number of bytes the spool file is configured to contain.

yourIntfClass

You must write this interface class. See Chapter 4, “Developing the Configuration Tool Communication Object,” for procedures and examples.

The configuration tool uses **yourIntfClass** to interface to a target system. This class provides methods that allow the configuration tool to determine what functionality this communication object supports, methods to handle system-specific configuration tool command-line parameters, and methods to open a view into the target system’s name space.

yourIntfClass Methods

abortNameSpaceLoad	Aborts the current name-space load in progress.
consumeOptions	Processes command-line arguments.
getChildren	Returns the child node names as a Tcl list.
list2path	Converts a Tcl list to a path string.
loadNameSpace	Loads the target system’s name space.
open	Loads configuration information and prepares the object for use.
options	Gets or sets command-line option keywords.
path2list	Converts a path string to a Tcl list.
selectionProcedures	Retrieves, deletes, executes and creates selection procedures.
supports	Indicates whether or not a feature is supported.
usage	Returns command-line usage information.
writeNameSpace	Saves the currently loaded name space to a file.

yourIntfClass::abortNameSpaceLoad

abortNameSpaceLoad

Return Value

No return value needs to be defined.

Remarks

Your **abortNameSpaceLoad** method must stop the currently active name space load activity.

If the **loadNameSpace** method is defined, this method must also be defined. If the **loadNameSpace** method is not defined, you do not need to define this method.

The configuration tool invokes this method when the user presses the Cancel push button on the Name-Space Load Status window. The configuration tool automatically created the Name-Space Load Status window when it invoked the **loadNameSpace** method.

yourIntfClass::consumeOptions

consumeOptions *var_name*

Return Value

No return value needs to be defined.

Parameters

var_name The name of the variable containing the list of command-line options.

Remarks

Your **consumeOptions** method must parse the global variable *$var_name* for command-line options specific to this class and remove any that are found.

Use the **utlPeekArg** and **utlGetArg** utilities to parse and remove recognized command-line options.

The configuration tool invokes this method to allow the communication object to remove any command-line options it is interested in.

yourIntfClass::getChildren

getChildren [**-filter** *pattern*] *node*

Return Value — Must return the names of children nodes for the target system's name space as a Tcl list.

Parameters — **-filter** *pattern* Only return names of children that match *pattern*.

node Full path name to the desired parent node, expressed as a Tcl list.

Remarks — Your **getChildren** method must get the names of the children under *$node*. The returned list of child names must only include each child's name and not any path information.

By definition, the root node of the name-space tree is the empty string. Therefore, *[$yourIntf getChildren ""]* returns the names of all nodes lying directly under the root node.

The configuration tool invokes this method whenever it needs to acquire the names of children nodes. For a given parent node, if the *dynamic name space* flag in the **supports** method is set to 0, this method will be invoked only once. Alternately, if the *dynamic name space* flag is set to 1, this method may be invoked more than once to check for new and deleted nodes.

yourIntfClass::list2path

list2path *list*

Return Value — Must return the human-readable version of *$list*.

Parameters — *list* A Tcl list that specifies a path in the logical name space.

Remarks — Your **list2path** method must convert *$list* into something that makes sense for your end users.

This method is the opposite of **path2list**.

The configuration tool invokes this method to format paths expressed as Tcl lists into syntax more easily understood by the user. Usually, this more easily understood syntax is simply the native syntax used for paths in the target system.

See Also "yourIntfClass::path2list" on page 7-41

---

yourIntfClass::loadNameSpace

loadNameSpace [**-command** *cmd*] [**-statusCommand** *scmd*]

Return Value No return value needs to be defined.

Parameters **-command** *cmd* The Tcl command to execute after the transfer is complete.

-statusCommand *scmd* The Tcl command to periodically execute while the transfer is in progress.

Remarks Your **loadNameSpace** method must start the transfer of name-space data from the target system to the configuration tool, then return without waiting for the transfer to complete. Your **writeNameSpace** method should save this name-space data to a file.

If you do not need to explicitly load name-space data, you can disable the name-space loading functionality for the target system by setting the *name space loading* flag in the **supports** method to 0. If you disable name-space loading in the **supports** method, you do not need to define this method.

Follow these guidelines:

- *scmd* should be run periodically by this communication object as the load progresses.
- *cmd* must be run by this communication object once the transfer has completed.
- All occurrences of %message in *scmd* must be replaced by the status message just prior to the execution of *scmd*.

- All occurrences of %error in *cmd* must be replaced by 0 if no errors occurred and 1 if errors occurred, just prior to the execution of cmd. If errors occurred, all occurrences of %message in *cmd* must be replaced by the error message just prior to the execution of *cmd*.

The configuration tool invokes this method when the user chooses the Load Name-space menu item from the main window's Control menu. The configuration tool creates and displays a Status window and arranges for this window to update whenever *scmd* and *cmd* execute.

See Also

"yourIntfClass::supports" on page 6-37 and "yourIntfClass::writeNameSpace" on page 7-45.

---

yourIntfClass::open

open [**-configDir** *name*] [**-debug**] [**-stub**][**-verbose**]

Return Value

No return value needs to be defined.

Parameters

-configDir *name* Set the configuration directory to *name*, which is the full directory path name to the current object's configuration g23.

-debug Print debug messages to stderr. This option can be useful to communication object developers for debugging.

-stub Fake the connection to the target system and generate some plausible name-space information. This can be useful for testing.

-verbose Print status messages to stderr. This option can be useful to end users for debugging.

Remarks

Your **open** method must open a connection to the target system.

The configuration tool invokes this method to open a connection to the target system. This connection will be used to obtain name-space information.

yourIntfClass::options

options *optkeys*

Return Value — Must return the current command-line option keyword names as a Tcl list.

Parameters — *optkeys* New names for command-line option keywords as a Tcl list.

Remarks — Your **options** method must get and set the command-line option keywords supported by this object. If the *$optkeys* parameter is not supplied, this method must return the list of currently supported command-line option keywords. If *$optkeys* is supplied, this method must set the keywords used for command-line options. In this case, the number of items in *$optkeys* must exactly match those in the returned list.

The configuration tool invokes this method to determine what command-line option keywords are currently supported and also to assign new command-line option keywords when there are keyword collisions with either the configuration tool's command-line option keywords or some other communication object's command-line option keywords.

yourIntfClass::path2list

path2list *path*

Return Value — Must return the Tcl version of *$path*.

Parameters — *path* The path to convert to a Tcl list.

Remarks — Your **path2list** method must convert *$path* to a Tcl list. This method must implement exactly the opposite conversions that are implemented in the **list2path** method.

The configuration tool invokes this method to convert paths expressed with a syntax more easily understood by the user to paths expressed as Tcl lists.

See Also — "yourIntfClass::list2path" on page 7-38

yourIntfClass::selectionProcedures

selectionProcedures

selectionProcedures *procedureName*

selectionProcedures *procedureName arguments*

selectionProcedures *procedureName arguments body apiVersion*

Return Value

The **selectionProcedures** method without parameters must return the names of all available selection procedures as a Tcl list of zero or more three-item lists:

```
{<procedureName> <arguments> <apiVersion>} {<procedureName> <arguments>
<apiVersion>} ...
```

The Tcl list should include selection procedures originating in both the communication object and the data server.

The **selectionProcedures** method with only the *procedureName* parameter must return a Tcl list of the names of deleted procedures:

```
<procedureName> <procedureName> ...
```

The **selectionProcedures** method with the *procedureName* and *arguments* parameters must return the selection procedures invoked, error messages encountered, and the selected sources as a Tcl list of zero or more two-item lists:

```
{{<procedureName> <arguments>} {errCode} {<selectedSources>}}
{{<procedureName> <arguments>} {errCode} {<selectedSources>}} ...
```

The **selectionProcedures** method with the *procedureName*, *arguments*, *body* and *apiVersion* parameters returns a Tcl list of the newly created selection procedures:

```
<procedureName> <procedureName> ...
```

Parameters

procedureName The name of the selection procedure.

arguments The arguments used by the selection procedure.

body The body text defining the selection procedure.

apiVersion The application programming interface (API) version of arguments and the values they return. In the E.02.20 version of Enterprise Link, the API version is 1.

Remarks

The **selectionProcedures** method is invoked by the configuration tool in order to query, create, execute, and delete selection procedures.

If only procedure names are supplied as a parameter, this method deletes selection procedures. If procedure names and arguments are supplied, this method executes selection procedures. If procedure names, procedure arguments, procedure bodies, and API versions are supplied, this method creates selection procedures.

---

yourIntfClass::supports

supports *receive data* | *transmit data* | *access* | *trigger* | *trigger focus* | *receive spooling* | *transmit spooling* | *dynamic name space* | *name space loading* | *selection procedures*

Return Value

If a parameter is not supplied, the supports method must return a list of all parameters that are potentially supported. If a parameter is supplied, the supports method must return 1 if the parameter is supported and 0 if it is not.

Parameters

- *receive data* The flow of data from the target system to the data server. A return value of 0 disables the Edit Method window's Direction/system 1-to-system 2 radio button.
- *transmit data* The flow of data from the data server to the target system. A return value of 0 disables the Edit Method window's Direction/system 2-to-system 1 radio button.
- *access* The display and editing of access information. A return value of 1 adds an "Access" menu item to the main window's Edit menu. Whenever this menu item is chosen, the Access window is displayed via the *yourIntfAccessGui* procedure.
- *trigger* The display and editing of system-specific trigger information. A return value of 1 adds a system-specific Trigger panel to the Trigger window.
- *trigger focus* An Edit Mapping window focus policy. A return value of 1 allows a branch in the appropriate Edit Mapping window to be shown in a highlighted color.

- *receive spooling* The spooling of data received from the target system. A return value of 0 disables a spooling item on the main window's Edit/Spooling cascade menu.
- *transmit spooling* The spooling of data sent from the data server to the target system. A return value of 0 disables a spooling item on the main window's Edit/Spooling cascade menu.
- *dynamic name space* A changing (dynamic) name space. A return value of 1 causes the configuration tool to check for new and deleted nodes when getting the children of a node.
- *name space loading* The manual loading of name-space information. A return value of 1 adds a "Load Name-space" menu item to the main window's Control menu. Whenever this menu item is chosen, the **loadNameSpace** method is invoked.
- *selection procedures* Selection procedures. A return value of 0 dims the Select checkbox in the Edit Mapping window.

Remarks

Your **supports** method should return 0 if the functionality specified by each parameter is unrecognized. The configuration tool invokes this method to determine the facilities provided by this communication object.

---

yourIntfClass::usage

usage

Return Value

Must return a list of Tcl lists: {{arg1summary arg1details} ...}

Remarks

Your **usage** method must return a list containing command-line argument usage information.

The configuration tool invokes this method when it needs to print a usage message due to command-line syntax errors.

yourIntfClass::writeNameSpace

writeNameSpace

Return Value — No return value needs to be defined.

Remarks — Your **writeNameSpace** method must save the currently loaded name space to a file.

The **loadNameSpace** method invokes this method.

8

Utility Reference

Utility Reference

This chapter documents the Enterprise Link utilities. These utilities provide routines that simplify developing the configuration tool and data server communication objects.

Note

The return value for many of the methods in this chapter is “No return value is defined.” Without an explicit “return” statement, a Tcl function returns whatever value happened to be returned by the last command used in the function. This means that no particular return value is defined: the method does not consistently return anything. Your code should not rely on any values returned from a method with no defined return value, since the method may return some different value in a future release of Enterprise Link.

Utility Name	Description	Page
utlClicksPerMilliSecond	Computes the number of clock clicks per millisecond for the host machine.	8-25
utlClose	Closes files.	8-55
utlCompleteEntryMsg	Temporarily displays an informative message in the entry widget.	8-27
utlCompleteGui	Creates, configures, and displays the completion window.	8-28
utlCurrentTime	Retrieves the current date and time for a specified time zone.	8-30
utlDecrypt	Decrypts encrypted character strings.	8-33
utlEncrypt	Encrypts character strings.	8-33
utlEnvVarName	Allows platform-independent access to environment variable names.	8-34
utlExitHan	Manages Tcl scripts that are called when the target system exits.	8-35
utlFileCopy	Copies contiguous blocks of bytes from one region of a file to another.	8-36
utlFilter	Filters *list*, removing any items not matching *pattern*.	8-37
utlFnameToStr	Converts a file name to a string.	8-38
utlFocusTraversal	Arranges for the current input focus to traverse the supplied list of Tk widgets.	8-39
utlFormatTime	Generates a formatted date and time string.	8-40
utlGetArg	Processes Tcl procedure parameters.	8-43
utlGetArray	Extracts a value from a nested array.	8-46
utlHelpGui	Creates, configures, and displays the help window.	8-47
utlIdleCursor	Restores the mouse cursor back to its original sprite.	8-17
utlIsNull	Determines if *variable* is empty.	8-48
utlIsPanel	Determines if *panel* is a valid Tk control panel.	8-49
utlJoinPathVar	Converts a Tcl list of paths into a single path value that can be assigned to the path environment variable.	8-50
utlList2Path	Converts a Tcl list to a path specification.	8-51
utlMkPanelEpilogue	Finishes window construction.	8-52
utlMkPanelPrologue	Starts window construction.	8-52

Utility Name	Description	Page
utlMkPanelVisible	Makes an existing window visible to the end user.	8-52
utlNls	Returns a localized string.	8-54
utlOpen	Opens files.	8-55
utlPath2List	Converts a path specification to a Tcl list.	8-57
utlPathVarSeparator	Returns the character used to separate the components of a path environment variable.	8-58
utlPattern2RegExp	Converts a pattern into a regular expression.	8-59
utlPeekArg	Parses the calling procedure's *args* variable.	8-43
utlPrepareWidget	Prepares the specified widgets for display.	8-60
utlPrintArray	Prints multi-dimensional array variables to an open file descriptor.	8-62
utlSetArray	Sets a value in a nested array.	8-63
utlSharedLibVarName	Returns the name of the environment variable used for shared library search paths on the host platform.	8-65
utlShiftTimeZone	Converts a date/time string from one time zone to another.	8-66
utlSplitPathVar	Converts the value of a path environment variable into a Tcl list of paths.	8-68
utlStrAlign	Pads the supplied strings with space characters.	8-69
utlStrToFname	Converts a string to a file name.	8-70
utlTableHeader	Initiates table construction.	8-71
utlTablePut	Prints the table.	8-71
utlTableRow	Adds one row of data to the table.	8-71
utlTimerNextTimeout	Finds the next timeout date.	8-73
utlTimerQuery	Retrieves timer information.	8-76
utlTimerStart	Starts a one-shot or periodic timer.	8-77
utlTimerStop	Stops the specified timer.	8-79
utlUnsetArray	Deletes multi-dimensional array variables.	8-80
utlWidgetState	Enables and disables Tk widgets.	8-81

Utility Name	Description	Page
utlWidgetText	Returns the user-visible text that identifies the specified widget.	8-82
yourIntfCompletionList	Composes a list of completions.	8-83

elFindServer

Finds a running data server.

Synopsis

elFindServer *service*

Description

elFindServer finds a running data server and returns a file descriptor for a dp_RPC connection to the data server. The data server's port name or number is configured to be *service*. A Tcl error is raised if no such server is running.

The *service* parameter is the service name or port number assigned to the data server in the configuration tool's Data Server Configuration window.

Return Value

No return value is defined.

elLeaveObjectHan

Manages configuration tool Tcl scripts that are called when the current configured object is no longer being edited.

Synopsis

elLeaveObjectHan {**-delete** *id*} *id command*

Description

elLeaveObjectHan manages Tcl scripts that are called when the current configured object is no longer being edited. It can add new handler scripts, delete existing handler scripts, return an existing handler script, or execute all defined handler scripts.

If **elLeaveObjectHan** is called without any parameters, it executes all currently defined handler scripts.

If the **-delete** option and *id* parameter are supplied, it deletes the handler script associated with this *id*.

If only the *id* parameter is suppled, it returns the handler script associated with this *id*.

If both the *id* and *command* parameters are supplied, it defines the Tcl script *command* as a new handler script, replacing any that existed with the same *id*.

Return Value

Returns the handler script associated with this *id* if only the *id* parameter is supplied. Otherwise, no return value is defined.

elLinkTriggerTimeout

Prepares to handle timer timeout events for *$methodName*.

Synopsis **triggerTimeout** *methodName prefix*

Return Value No return value is defined.

Parameters *methodName* The name of the method to trigger.

prefix The trigger variable prefix string.

Remarks Prepares to handle timer timeout events for *$methodName*. When a timeout occurs, the **elLinkClass::execute** method is invoked.

The trigger configuration variables needed by this routine can be defined by sourcing a trigger configuration file (*.trig). The names of these variables must begin with the string specified by *prefix*.

elNSpaceGui

Creates, configures and displays the Configuration Tool's Edit Name Space window utility.

Synopsis

elNSpaceGui [-attrCompletionProc *name*]**[-attrDefaultValueProc** *name*]**[-applyProc** *name*]**[-revConvertProc** *name*]**[-attrNames** *names*]
[-attrCompletionStyle *style*]**[-attrPrintFormats** *fmts*]**[-fileName** *file_name*]**[fileNameProc** *name*]**[-helpOnAttrsText** *text*]
[-helpOnAttrsWindowText *text*]**[-helpOnWindowText** *text*]
[-loadNameSpaceCmd *name*]**[-prefix** *prefix*]
[-title *title*]**[-unotesFileName** *file_name*] *panelName*

Description

Creates, configures and displays the Configuration Tool's Edit Name Space dialog window. This procedure provides communication object developers with a ready-made Name Space Editor. End users use this editor to add, modify and delete branches of the name space, as well as to modify name space attribute values. The Name Space Editor contains two important windows:

- Edit Name Space window. Branches of the name space are added, modified and deleted using this window. This is the window initially displayed when elNSpaceGui is called.
- Edit Name Space Attributes window. Attribute values are modified using the Edit Name Space Attributes window. This window can be displayed by choosing Attributes... from the Edit Name Space window's Edit menu. Attributes are displayed as a two column table, with attribute names on the left, and corresponding attribute values on the right. The end user must complete four steps to modify the value of an address's attribute. First, the desired address must be selected on the Edit Name Space window. The second step is to select the desired attribute on the Edit Name Space Attributes window. Then, the displayed attribute value can be edited to the new value. The final step is to press the Modify push button.

The *panelName* parameter specifies a name to use for the Edit Name Space window instance. It is best if the panel names are unique and have some reference to the communication objects. This can be done by embedding the name of the communication object in the panel name, for example:
`.edit_SAP_nspace`

Options

The -**attrNames** *names* option is used to specify the names of the attributes associated with a name space. -**attrNames** *names* is a TCL list of one or more attribute names. Attribute names are displayed in the Edit Name Space Attributes window in the same order as they appear in the TCL list. The following example specifies the attributes “myAttr1”, “myAttr2’, and ‘myAttr3”.

```
-attrNames [list “myAttr1” “myAttr2” “myAttr3” ]
```

If the -**attrNames** *names* option is omitted, the Edit Name Space utility assumes the name space has no attributes, and thus does not provide an Edit Name Space Attributes window.

The -**attrCompletionProc** *name* option is used to specify the name of a procedure that returns the possible attribute value completions. These value completions are for text typed into the Edit Name Space Attribute window’s attribute value text editor. The -**attrCompletionProc** *name* procedure is called whenever the end user presses the ESC key in the attribute value text editor. The synopsis for this user-written procedure is shown below:

```
[name] [address] [attr_name]
where:
[name] is the procedure’s name
[address] is the currently selected address expressed as a TCL list
[attr_name] name of the currently selected attribute
```

The procedure is expected to return a TCL list of zero or more possible attribute values.

The -**attrCompletionStyle** *style* option is used to specify the text completion behavior of the attribute value text editor in the Edit Name Space Attributes window. -**attrCompletionStyle** *style* is a TCL list of the following two keywords: NORMAL and SUFFIX. Attributes whose values are addresses should use the SUFFIX keyword. For other attributes, the NORMAL keyword should be used. There should be one item in the -**attrCompletionStyle** *style* list for each related item in the *names* list supplied to the -**attrName**s *names* option.

The -**attrDefaultValueProc** *name* option is used to specify the name of a procedure that returns the default value for a specified address and attribute. This procedure is called whenever the end user creates new addresses. The synopsis for this user-written procedure is shown below:

```
[name] [address] [attr_name]
where:
[name] is the procedure's name
[address] is the currently selected address expressed as a TCL list
[attr_name] name of the currently selected attribute
```

The procedure is expected to return a default value for the supplied address and attribute.

The -**applyProc** *name* option is used to specify the name of a procedure that is invoked whenever the Edit Name Space window's Apply push button is pressed. For example, this hook allows your communication object to convert the configured name space in it's new layout as it is currently displayed in the Edit Name Space window. The *name* procedure updates the logical name space displayed in the Edit Mapping window. The synopsis for this user-written procedure is shown below:

```
[name]
where:
[name] is the procedure's name
```

The value returned by this procedure is ignored.

The **-revConvertProc** *name* option is used to specify the name of a procedure that converts name space configuration files from an old format which was shipped in a previous release of the product, to the current format. This procedure is called whenever the name space configuration file is loaded. The synopsis for this user-written procedure is shown below:

```
[name]
where:
[name] is the procedure's name
```

The value returned by this procedure is ignored.

The -**loadNameSpaceCmd** *name* option is used to load name space data from the Communication object. The TCL command *name* is executed whenever the Edit Name Space window's Load Name Space command is chosen from the Control menu. (See "Developing the loadNameSpace Method" on page 4-18).

The -**title** *title* option is used to specify the title of the Edit Name Space window. You can give this window a title that will reflect what the end user is doing in that window. Set up the *title* like this example:

```
"my -- Edit Name Space"
```

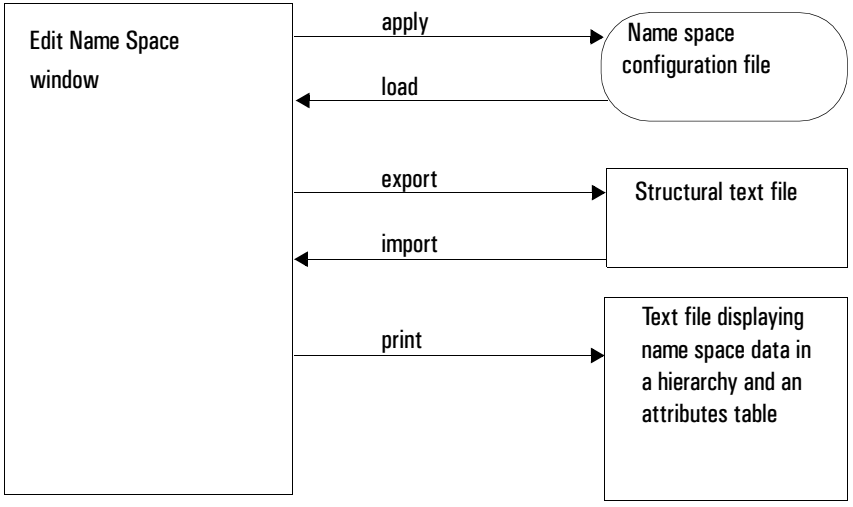
Or, if you want NLS support, set it up like this example:

```
[utlNls "my -- Edit Name Space"]
```

The -**fileName** *file_name* option is used to specify the name of the configuration file to which name space data is stored when the end user presses the apply push button. The configuration file is part of the current configuration object. Set up the *file_name* like this example:

```
my_namespace_file
```

If you wish to store the configuration file in some directory other than the current configured object's configuration directory, specify that directory. If a directory path is specified, that will be where the name space data is stored. In the diagram below, the name space configuration file is where the name space data is stored.



The -**fileNameProc** *name* option is executed whenever you need to refer to the name of the configuration file. -**fileNameProc** *name* overrides the -**fileName** *name* option. The procedure *name* is called as shown below:

```
[name]
where:
[name] is the procedure's name
```

name is expected to return a full file path name as a character string.

The -**attrPrintFormats** *fmts* option is used to specify how to format attribute values when printing them. -**attrPrintFormats** *fmts* is a TCL list of format specifications. Formatted values will be displayed in a text file specified by the end user. A format must be assigned to every attribute name given from the -**attrNames** *name* list.

This option is usually used to restrict the width of values that are many characters long. Since attribute values are displayed in a table, the table's columns must be formatted so that each column is reasonably wide enough for the end user to view the values easily.

Each list-item can either be:

Specification	Layout of print format	Example
An empty TCL list	No column width limit. Columns can be as wide as the widest value.	[ ]
A positive integer number	Column width is restricted to the maximum width that you specify. Values wider than this limit will be broken into multiple lines for display purposes.	8
Two positive integer numbers in the form of a TCL list: { < soft limit > < hard limit > }	Column width is specified by both soft and hard limits. Attribute values will be broken into multiple lines at white space characters to achieve the soft limit, but if there is no appropriate white space, the hard limit will be applied.	{20 30}

The keyword "SETTINGS"	Settings of token oriented sequences will be broken into multiple lines so that there is one token/flag per line. A token oriented sequence might be:
	`my_token=<value> token2=<value2> flag1`
	In multiple lines these sequences become:
	`my_token=<value>`
	`token2=<value2>`
	`flag1`
		SETTINGS

The -**helpOnWindowText** *text*, -**helpOnAttrsWindowText** *text* and the -**helpOnAttrsText** *text* options are used to specify the raw text for three particular online Help files in the Name Space Editor. Although each window in the Name Space Editor has a Help menu, the two windows that these three options apply to are the:

1. Edit Name Space window.
2. Edit Name Space Attributes window.

The **-helpOnWindowText** *text* option identifies the online Help text for the Edit Name Space window. This help text describes how to use the Edit Name Space window.

The -**helpOnAttrsWindowText** *text* option identifies the online Help text for the Edit Name Space Attributes window.

The -**helpOnAttrsText** *text* option identifies the online Help text for the attributes on the Edit Name Space Attributes window. This text describes the meaning of each attribute in the Edit Name Space Attributes window.

The -**unotesfileName** *file_name* option is used to specify the name of the file in which the end user's notes for the online help files will be stored. The end users can write their own notes for later viewing in the user notes area on the online Help windows. An example of how to specify the *file_name* option is:

```
my_usernotes_file
```

The -**prefix** *prefix* option is used to specify which data structures will hold the Name Space Editor. Specify the initial letters of your communication object. By doing this, you are attaching any changes made to the name space, to your communication object. When the name space structure is edited in the Edit Name Space window, the data structure changes as well. The TCL array variables created and managed by the Edit Name Space window are:

```
<prefix>nspace_tree
<prefix>nspace_attributes
```

The double dash (- -) option marks the end of options. The double dash (- -) removes the special meaning of the parameter that follows it. For example, for the function "create":

```
create -- -file
```

Placed in front of `-file`, the double dash make sure that `-file` can be separated from it's special meaning. The argument following a (- -) will be treated as character string even if it starts with a (-).

Return Value

The return value for **elNSpaceGui** is undefined.

See also

"utlNls" on page 8-54.

Example

The following example brings up a name space editor for the hypothetical "MY" communication object:

```
MYAttrCompletionProc {address attr_name} {
set items [compose list of possible attribute values here]
return $items
}
MYAttrDefaultValueProc {address attr_name} {
return [default value for given address and attribute name]
}
MYApplyProc {} {
[update any caller-created windows that display the current name space]
}
MYNSpaceCfgConvertProc {} {uplevel 1 {
# --- fetch the config-file's revision number ---
upvar 0 ${pre}nspace_config_rev config_rev
# --- convert rev 1 files to rev 2 ---
if {$config_rev == 1} {
[convert configuration rev 1 to rev 2 here]
set config_rev 2
}
# --- convert rev 2 files to rev 3 ---
if {$config_rev == 2} {
[convert configuration rev 2 to rev 3 here]
set config_rev 3
}
}}
```

utlAbsPath

Converts any file path specification into an absolute file path specification.

Synopsis

utlAbsPath *path dir*

Description

utlAbsPath determines the absolute path denoted by *path* relative to *dir*. If *dir* is omitted, the directory returned by the Tcl command *[pwd]* is used.

Return Value

Returns the full (absolute) file path name.

utlBusyCursor, utlIdleCursor

Temporarily changes the sprite used for the mouse cursor to an hourglass.

Synopsis

utlBusyCursor *level*

utlIdleCursor [**-force**]

Description

utlBusyCursor changes the cursor to an hourglass sprite, while **utlIdleCursor** restores the mouse cursor back to its original sprite.

Calls to **utlBusyCursor** and **utlIdleCursor** can be nested. Passing **utlIdleCursor** the **-force** option collapses all nesting and forces the cursor back to its original sprite. Passing **utlBusyCursor** the *level* parameter sets the resulting nesting level to *level*.

Whenever you need to temporarily set a busy cursor back to its original sprite, call **utlIdleCursor -force** and note the return value. Later, when you are ready to restore the cursor back to an hourglass sprite, call **utlBusyCursor** *level* using the value returned by the previous call to **utlIdleCursor** for *level*.

Return Value

utlIdleCursor returns the busy-cursor nesting level in effect just before this call.

utlBusyCursor returns the busy-cursor nesting level in effect just after this call.

Caution

The maximum time that the mouse cursor remains an hourglass sprite is 90 seconds. After 90 seconds, the cursor automatically reverts back to its original sprite and the busy-cursor nesting level collapses to zero.

Errors that occur while the mouse cursor is displayed as an hourglass sprite cause the cursor to revert back to its original sprite and the busy-cursor nesting level to collapse to zero. This is done in the utility-library-supplied error handler procedure.

All calls to **utlIdleCursor** while the mouse cursor is not busy are ignored.

Example

```
# start the slow task
utlBusyCursor
< start doing your time-consuming task here >

# ask the end-user a question along the way
set orig_nesting [utlIdleCursor -force]
< do an interactive query with the end-user here >
utlBusyCursor $orig_nesting

< do more time consuming stuff here >

# slow task is done!
utlIdleCursor
```

utlCanonicalizeList

Converts a Tcl list into canonical (also known as *standard*) form.

Synopsis **utlCanonicalizeList** *name*

Description Finds the standard representation for a named list in the caller's context. For example, extra white space between items is removed and quoted items are converted to brace-delimited items.

Return Value The canonical (standard) representation of the list.

utlChkAlpha

Checks the validity of *string*, generating an error if *string* is empty or invalid.

Synopsis

utlChkAlpha [**-comment** *com*] [**-expected** *exp*] *string err_text*

Description

utlChkAlpha checks the validity of *string*, generating an error if it detects one of the following:

- *string* is empty.
- *string* does not match one of the items in the list <*exp*>.

The *err_text* parameter is text that identifies which entry widget on a window contains the erroneous value and is usually the text returned by a call to **utlWidgetText**.

If the **-comment** *com* option is supplied, *com* replaces the initial error text in the generated error message.

If the **-expected** *exp* option is supplied, *string* must exactly match one of the items in the Tcl list specified by *exp*.

Return Value

No return value is defined.

See Also

"utlChkInt" on page 8-23, "utlChkName" on page 8-24, "utlChkCfgFileRev" on page 8-21, and "utlWidgetText" on page 8-82.

utlChkCfgFileRev

Checks the revision of the just sourced configuration file.

Synopsis

utlChkCfgFileRev *rev_var_name expected_rev fname*

Description

utlChkCfgFileRev checks the revision of the just-sourced configuration file, generating an error if it detects one of the following:

- The variable specified by *rev_var_name* does not exist.
- The value of the variable specified by *rev_var_name* does not equal *expected_rev*.

The *rev_var_name* parameter is the name of the variable containing the configuration file's revision number.

The *expected_rev* parameter is the expected value of the variable specified by *rev_var_name*.

The *fname* parameter is the full path name of the configuration file just sourced.

Return Value

No return value is defined.

Example

The following example loads a configuration file by sourcing it. Once the file is sourced, this utility checks that the variable *my_config_rev* was created and that it has the right value.

```
proc myCfgLoad {} {

global el_app_cfg_subdir \
       el_app_obj_name \
       my_cfg_file_name \
       my_config_app_rev \
       my_password \
       my_user_name

   #  compose file name
   set fname \
      "${el_app_obj_name}${el_app_cfg_subdir}/$my_cfg_file_name"

   #  if the specified config file does not exist or is empty
   if {([file exists $fname] != 1) || ([file size $fname] == 0)} {
      < assign default values to `my_password' and 'my_user_name'
      here >
      return
   }

   #  load the configuration
   set _prefix "my_"
   source $fname

   utlChkCfgFileRev “my_config_rev” $my_config_app_rev $fname
}
```

See Also

“utlChkAlpha” on page 8-20, “utlChkInt” on page 8-23, and “utlChkName” on page 8-24.

utlChkInt

Checks the validity of *number*, generating an error if *number* is invalid.

Synopsis

utlChkInt [**-comment** *com*] [**-min** *num*] [**-max** *num*] *number err_text*

Description

utlChkInt checks the validity of *number*, generating an error if it detects one of the following:

- *number* is empty.
- *number* is not an integer numeric value.
- *number* is greater than *max*.
- *number* is less than *min*.

err_text is text that identifies which entry widget on a window contains the erroneous value and is usually the text returned by a call to **utlWidgetText**.

If the **-comment** *com* option is supplied, *com* replaces the initial error text in the generated error message.

If the **-min** *num* option is supplied, *num* specifies the minimum valid value for *number*.

If the **-max** *num* option is supplied, *num* specifies the maximum valid value for *number*.

Return Value

No return value is defined.

See Also

"utlChkAlpha" on page 8-20, "utlChkName" on page 8-24, "utlChkCfgFileRev" on page 8-21, and "utlWidgetText" on page 8-82.

utlChkName

Checks the validity of *name*, generating an error if *name* is not a valid Tcl variable name.

Synopsis

utlChkName [**-comment** *com*] *name err_text*

Description

utlChkName checks the validity of *name*, generating an error if it detects one of the following:

- *name* is empty.
- *name* does not begin with an alphabetic character.
- *name* contains nonprinting characters.

The *err_text* parameter is text that identifies which entry widget on a window contains the erroneous value and is usually the text returned by a call to **utlWidgetText**.

If the **-comment** *com* option is supplied, *com* replaces the initial error text in the generated error message.

Return Value

No return value is defined.

See Also

"utlChkAlpha" on page 8-20, "utlChkInt" on page 8-23, "utlChkCfgFileRev" on page 8-21, and "utlWidgetText" on page 8-82.

utlClicksPerMilliSecond

Computes the number of clock clicks per millisecond for the host machine.

Synopsis **utlClicksPerMilliSecond**

Description Computes the number of clock clicks per millisecond for the host machine.

Return Value The number of clock clicks per millisecond.

utlCompleteEntry

Handles completion for Tk entry widgets.

Synopsis

utlCompleteEntry [**-errScheme**] [**-title** *title*] [**-wait**]
panel widget choices

Description

utlCompleteEntry handles completion for Tk entry widgets. It completes the entered value if unique, otherwise it displays a list of matching completions. This utility is a wrapper around the utility **utlCompleteGui**.

The **-errScheme** option causes the completion window to use the “error” color scheme.

The **-title** *title* option sets the title for the completion window to *title*.

The -**wait** option causes **utlCompleteEntry** to return only when an item has been selected from the completion list. If there is only one possible completion, **utlCompleteEntry** returns immediately.

The *panel* parameter specifies the name to assign to the completion window's top-level window.

The *widget* parameter specifies the existing Tk entry widget to perform completion on.

The *choices* parameter is a Tcl list of one or more possible completions. For this list, items containing reverse-solidus characters (\) must be enclosed in braces (both the Tcl **lappend** and Tcl **list** commands do this automatically).

Return Value

No return value is defined.

See Also

“utlCompleteEntryMsg” on page 8-27 and “utlCompleteGui” on page 8-28.

utlCompleteEntryMsg

Temporarily displays an informative message in the entry widget.

Synopsis

utlCompleteEntryMsg *panel widget message*

Description

utlCompleteEntryMsg temporarily displays an informative message in the entry widget. This message is typically an error message describing why completion could not be performed. The global variable *utl_complete_mom_msg_time* specifies, in milliseconds, how long to display the message. If *utl_complete_mom_msg_time* is not set the first time that either **utlCompleteGui**, **utlCompleteEntry**, or **utlCompleteEntryMsg** are called, *utl_complete_mom_msg_time* defaults to 375 milliseconds.

The *panel* parameter specifies the name to assign to the completion window's top-level window.

The *widget* parameter specifies the existing Tk entry widget to perform completion on.

The *message* parameter is a text string describing why completion was denied.

Return Value

No return value is defined.

See Also

"utlCompleteEntry" on page 8-26 and "utlCompleteGui" on page 8-28.

utlCompleteGui

Creates, configures, and displays the completion window.

Synopsis

utlCompleteGui [**-command** *script*] [**-title** *title*] [**-errScheme**] [**-okCommand** *ok_script*] [**-variable** *var_name*] [**-wait**] [**-suffix**] *panel choices*

Description

utlCompleteGui creates, configures, and displays the completion window. Completion windows display a list of completion choices to end users.

In typical use, the completion window pops up to display a list of possible selections, the end user then selects an item from this list and presses the OK push button. This causes the selected item to be written to *var_name*, and for *script* and *ok_script* to run. If nothing was selected when OK was pressed, *var_name* is only updated with the first *n* characters of what is common to all possible *choices*.

The **-command** *script* option causes *script* to execute whenever any completion occurs. All occurrences of %value in *script* are replaced by the final completion value selected in the Tk list widget just prior to the script's execution.

The **-errScheme** option causes the completion window to use the error color scheme.

The **-okCommand** *ok_script* option causes *ok_script* to execute when the OK push button is pressed. All occurrences of %value in *ok_script* are replaced by the final completion value selected in the Tk list widget just prior to the script's execution.

The **-title** *title* option sets the title for the completion window to *title*.

The **-variable** *var_name* option arranges for the variable specified by *var_name* to be set to the chosen selection.

The **-wait** option causes **utlCompleteGui** to only return once an item has been selected.

The -**suffix** option prevents **utlCompleteGui** from completing partially specified items. This option enables users to compose expressions and sentences, then append an item chosen from a completion list to that expression or sentence.

The *panel* parameter specifies the name to assign to the completion window's top-level window.

The *choices* parameter is a Tcl list of one or more possible completions. For this list, items containing reverse-solidus characters (\) must be enclosed in braces (both the Tcl **lappend** and Tcl **list** commands do this automatically).

Return Value

The return value is undefined if the **-wait** option is omitted, otherwise the chosen selection is returned.

See Also

"utlCompleteEntry" on page 8-26, "utlCompleteEntryMsg" on page 8-27, and "utlPrepareWidget" on page 8-60.

utlCurrentTime

Retrieves the current date and time for a specified time zone.

Synopsis

utlCurrentTime [-**zone** *zone*] *format*

Description

utlCurrentTime retrieves the current date and time for a specified time zone.

The -**zone** *zone* option allows you to specify a time zone for the returned date/time string. *zone* can be assigned two different types of values: a time zone name, or a full time zone specification. For a time zone name, some possible values supported on HP-UX systems are the following:

Value	Description
MET	(Middle European Time)
WET	(Western European Time)
GMT	(Greenwich Mean Time)
BST	(British Summer Time)
NST	(Newfoundland Standard Time)
NDT	(Newfoundland Daylight Time)
AST	(Atlantic Standard Time)
ADT	(Atlantic Daylight Time)
EST	(Eastern Standard Time)
EDT	(Eastern Daylight Time)
CST	(Central Standard Time)
CDT	(Central Daylight Time)
MST	(Mountain Standard Time)
MDT	(Mountain Daylight Time)
PST	(Pacific Standard Time)
PDT	(Pacific Daylight Time)
YST	(Yukon Standard Time)

Value	Description
YDT	(Yukon Daylight Time)
AST	(Aleutian Standard Time)
ADT	(Aleutian Daylight Time)

For a full time-zone specification, *zone* must have the following form:

```
[:]STDoffset[DST[offset][,rule]]
```

where

STD and DST are one or more bytes that designate the standard time zone (STD) and summer, or daylight savings time zone (DST). STD is required. If DST is not specified, summer time does not apply in this locale. Any characters other than the numerals 0 through 9, the comma (,) character, the minus (-) character, or the plus (+) character are allowed.

offset is the value that must be added to local time to arrive at Coordinated Universal Time (UTC). offset has the following form:

```
hh[:mm[:ss]]
```

where

the hour (hh) is any value from 0 through 23. The optional minutes (mm) and seconds (ss) fields are a value from 0 through 59. The hour field is required.

If offset is preceded by a minus (-) character, the time zone is east of the Prime Meridian. If offset is preceded by a plus (+) character, the time zone is west of the Prime Meridian. The default case is west of the Prime Meridian.

rule indicates when to change to and from daylight savings time. rule has the following form:

```
date/time,date/time
```

where

the first date/time specifies when to change from standard to daylight savings time, and the second date/time specifies when to change back. These fields are expressed in current local time. The form of date should be one of the following:

Dm.d Day of the month, where m is the month (1 through 12) and d is the day of the month (1 through 31).

Jn Julian day (1 through 365). Does not count leap days (February 29).

n The zero-based Julian day (0 through 365). Counts leap days (February 29).

Mm.n.d The day of the week of the month, where m is the month (1 through 12), n is the week of the month (1 through 5, with 1 being the week in which the first day of the month falls) and d is the day of the week (0 through 6, with 0 being Sunday).

time has the same format as offset except that no leading sign ("-" or "+") is allowed. The default, if time is not given, is 02:00:00.

If the -**zone** option is not specified, the local time zone is used.

format specifies the desired format for the returned date/time character string. **utlCurrentTime** supports the same date/time formats as the utility function **utlFormatTime**.

Return Value

Returns a formatted date/time string.

utlEncrypt, utlDecrypt

Encrypts or decrypts character strings.

Synopsis

utlEncrypt *key value*

utlDecrypt *key value*

Description

utlEncrypt encrypts user-entered passwords that need to be stored in configuration files. **utlDecrypt** decrypts user-entered passwords that were encrypted by **utlEncrypt**.

utlEncrypt encrypts the character string *value* using the encryption *key*. The *key* parameter is any arbitrary character string. The *value* parameter is the user-entered password.

utlDecrypt decrypts the cipher code *value* using the encryption *key*. To successfully decrypt *value*, the *key* parameter passed to **utlDecrypt** must match the *key* parameter originally used to generate the cipher code. The *value* parameter is the encrypted user-entered password.

Return Value

utlEncrypt returns the encrypted cipher code as a Tcl list.

utlDecrypt returns the decrypted clear-text character string for success and the passed-in cipher code for failure.

utlEnvVarName

Allows access to environment variable names independent of specific platforms.

Synopsis

utlEnvVarName *name*

Description

utlEnvVarName determines which string to use for references to the environment variable specified by *name*. If the environment variable *name* does not exist, **utlEnvVarName** searches for the first defined environment variable whose name differs from the requested name in case only, then uses that variable's name. If no such environment variable is defined, **utlEnvVarName** uses the requested name without modification.

utlEnvVarName is useful in Tcl scripts that must run on both UNIX and Windows NT systems, where there are differences in case-sensitivity.

Return Value

Returns the string that should be used to access the environment variable specified by *name*.

utlExitHan

Manages Tcl scripts that are called when the target system exits.

Synopsis

utlExitHan [**-delete** *id*] *id command*

Description

utlExitHan manages Tcl scripts that are called when the target system exits. It can add new handler scripts, delete existing handler scripts, return existing handler scripts, or execute all defined handler scripts.

If **utlExitHan** is called without any parameters, it executes all currently defined handler scripts.

If the *id* parameter is supplied, it returns the handler script associated with this *id*.

If the **-delete** option and the *id* parameter are supplied, it deletes the handler script associated with this *id*.

If both the *id* and *command* parameters are supplied, it defines the Tcl script **command** as a new handler script and replaces any that existed with the same *id*.

Return Value

Returns the handler script associated with this *id* if the *id* parameter is supplied. Otherwise, no return value is defined.

utlFileCopy

Copies contiguous blocks of bytes from one region of a file to another.

Synopsis

utlFileCopy *from_fd from_offset to_fd to_offset length*

Description

Copies the specified number of bytes from the offset specified in *from_offset* in the source file's open file descriptor to the offset specified in *to_offset* in the destination file's open file descriptor. This routine handles situations in which *to_fd* and *from_fd* refer to the same file and the source and destination regions overlap.

from_fd is the source file's open file descriptor.

from_offset is the offset in source file.

to_fd is the destination file's open file descriptor.

to_offset is the offset in destination file.

length is the number of bytes to copy.

Return Value

No return value is defined.

utlFilter

Filters *list*, removing any items not matching *pattern*.

Synopsis

utlFilter *list pattern*

Description

utlFilter filters *list*, removing any items not matching *pattern*.

The *list* parameter specifies the Tcl list to be filtered.

The *pattern* parameter specifies the pattern using pattern matching notation.

Return Value

Returns the items in *list* matching *pattern*, as a Tcl list.

utlFnameToStr

Converts a file name to a string.

Synopsis **utlFnameToStr** *name*

Description **utlFnameToStr** converts a file *name* to a string and restores characters that were encoded by **utlStrToFname**.

Return Value Returns the resulting string.

See Also "utlStrToFname" on page 8-70.

utlFocusTraversal

Arranges for the current input focus to traverse the supplied list of Tk widgets.

Synopsis

utlFocusTraversal [**-noLoop**] *widget1 widget2 widget3* ...

Description

utlFocusTraversal arranges for the current input focus to traverse the list of Tk widgets whenever the end user types the carriage return key. This is accomplished by binding a simple return-key handler to each widget. This handler sets the focus to the next widget in the supplied list. All supplied widgets should be Tk entry widgets.

By default the traversal is closed, meaning that the last widget in the supplied list advances the focus back to the first widget.

If the **-noLoop** option is supplied, no return-key handler will be associated with the last widget in the supplied list. Therefore, the focus will not advance beyond this widget.

The *widget1 widget2 widget3* ... parameters are the list of Tk widgets.

Return Value

No return value is defined.

See Also

"utlPrepareWidget" on page 8-60.

utlFormatTime

Generates a formatted date and time string.

Synopsis

utlFormatTime *format year month day hour minute second usec*

Description

utlFormatTime generates a formatted date and time string.

The *format* parameter specifies the desired date and time format. It may contain any of the following conversion specifiers:

Specifier	Description
y	year as a number (no leading zeros)
yy	year as a two-digit number
yyyy	year as a four-digit number
m	month as a number (no leading zeros)
mm	month as a two-digit number
mmm	abbreviated name of the month
mmmm	full name of the month
d	day of the month as a number (no leading zeros)
dd	day of the month as a two-digit number
ddd	abbreviated name for the day of the week
dddd	full name for the day of the week
H	hour as a number (no leading zeros)
HH	hour as a two-digit number
M	minute as a number (no leading zeros)
MM	minute as a two-digit number
S	second as a number (no leading zeros)
SS	second as a two-digit number
U	microseconds as a number (no leading zeros if integral)
UU	microseconds as a three-digit number
UUU	microseconds as a six-digit number

Specifier	Description
AM/PM	12-hour clock with an AM/PM indicator
am/pm	12-hour clock with an am/pm indicator

If the U, UU, or UUU conversion specifier is immediately preceded with either a period (.) or comma (,) character, the formatted result is the fractional number of seconds. Otherwise, the formatted result is the total number of microseconds.

If neither the AM/PM or am/pm conversion specifiers are present, all H and HH conversion specifiers use a 24-hour clock. Otherwise, they use a 12-hour clock.

If any of the listed conversion characters (y, m, d, H, M, S, or U) are to appear in *format* as literal characters, they must be quoted by preceding each of them with a reverse-solidus (\) character .

year specifies the year to include in the formatted result. For this parameter, 0-69 maps to 2000-2069 and 70-99 maps to 1970-1999.

month specifies the month of the *year* and is an integer from 1 to 12 (January to December).

day specifies the day of the *month* and is an integer from 1 to 31.

hour specifies the hour of the *day* and is an integer from 0 to 23.

minute specifies the minute of the *hour* and is an integer from 0 to 59.

second specifies the number of seconds into the current *minute* and is an integer from 0 to 60 (including leap-seconds).

usec specifies the number of microseconds into the current *second* and is an integer from 0 to 1000000.

The parameters *year*, *month*, *day*, *hour*, *minute*, *second*, and *usec* may each contain a literal string. Literal strings include negative integers, floating point numbers, and all nonnumeric values. Literal strings are inserted into the returned value, as is, at the positions specified by *format*. This is useful for wildcarding.

Return Value

A formatted date and time string.

Example

```
set s [utlFormatTime “yy-mm-dd HH:MM:SS” 96 03 20 7 47 22]

set s [utlFormatTime "yyyy/mm/dd HH:MM:SS.UUU" \
    96 03 20 7 47 22 500000]

set s [utlFormatTime "yyyy/mm/dd HH:MM:SS.UUU" 96 03 20 7 47 22.5]

set s [utlFormatTime "dddd, mmmm dd, yyyy H:MM:SS AM/PM" \
    96 03 20 7 47 22]

set s [utlFormatTime “\\year=yyyy \\month=mm \\da\\y=dd hour=HH \
    \\minute=MM secon\\d=SS usec=UU” 96 03 20 7 47 22]
```

Caution

The ability to pass values that are not positive integers for the parameters *year*, *month*, *day*, *hour*, *minute*, *second*, and *usec* can obscure parameter passing defects in your code.

utlGetArg, utlPeekArg, utlArgEnd

Processes Tcl procedure parameters.

Synopsis

utlGetArg *keyword default value*

utlPeekArg *keyword default value*

utlArgEnd

Description

utlGetArg, **utlPeekArg**, and **utlArgEnd** process Tcl procedure parameters.

utlArgEnd checks the calling procedure's *args* variable for unprocessed parameters and reports any errors found.

utlGetArg and **utlPeekArg** parse the calling procedure's *args* variable looking for a command-line parameter matching *keyword*. These routines handle three forms of procedure parameters:

- Optional procedure parameters with one argument such as:

  ```
  -color "white"
  ```

  In this case, **-color** is defined to be the keyword and *white* is the optional parameter's argument.

- Optional procedure parameters with no arguments such as: -rewind

  In this case, **-rewind** is the keyword and the optional parameter has no argument.

- Required procedure parameters.

For procedure parameters with one argument:

If *keyword* is supplied and found in *args*, and *default* is supplied, then the assumed value immediately following *keyword* is returned. Before returning, **utlGetArg** removes both *keyword* and its associated value from *args*. **utlPeekArg** never modifies the value of the calling procedure's *args* variable.

If *keyword* is not found in *args*, *default* is returned and *args* remains unchanged.

For procedure parameters with no argument (boolean return values):

If *keyword* is supplied and found in *args*, and *default* and *value* are not supplied, then no value is assumed to follow *keyword* and the value 1 is returned. Before returning, **utlGetArg** removes *keyword* from *args*. **utlPeekArg** never modifies the value of the calling procedure's *args* variable.

If *keyword* is not found in *args*, the value 0 is returned and *args* remains unchanged.

For procedure parameters with no argument (custom return values):

If *keyword* is supplied and found in *args*, and *default* and *value* are supplied, then no value is assumed to follow *keyword* and *value* is returned. Before returning, **utlGetArg** removes *keyword* from *args*. **utlPeekArg** never modifies the value of the calling procedure's *args* variable.

If *keyword* is not found in *args*, *default* is returned and *args* remains unchanged.

For required procedure parameters:

If *keyword*, *default*, and *value* are not supplied, then the assumption is that a required parameter is present and the first item in *args* is returned. Before returning, **utlGetArg** removes this item from *args*. **utlPeekArg** never modifies the value of *args,* therefore is not very useful with required procedure parameters.

The calling routine can demark the end of optional procedure parameters and the beginning of required procedure parameters with the "--" flag. All procedure parameters following this flag are treated as required parameters. This allows the calling routine to pass in required parameters whose values match any of the procedure's optional parameters.

Return Value

utlGetArg and **utlPeekArg** return either the value appropriate for the found keyword or the default value if the keyword was not found.

No return value for **utlArgEnd** is defined.

Example

```
# Usage: foo [-decrement] [-file <name>] [-rewind] string char

proc foo args {
   #  fetch optional parameters
   set inc [utlGetArg "-decrement" "1" "-1"]
   set fname [utlGetArg "-file" ""]
   set rewind [utlGetArg "-rewind"]

   #  fetch required parameters
   set string [utlGetArg]
   set char [utlGetArg]
   utlArgEnd

   #  your procedure body here...
   for {set i 0} {... } {incr i $inc} {... }

   if {$fname != ""} {set f [open $fname w]
      } else {set f stderr}

   if {$rewind} {... }
      :
      :
      :
}
```

utlGetArray

Extracts a value from a nested array.

Synopsis

utlGetArray *arrayName index1 ... indexN*

Description

utlGetArray extracts a value from a nested array. If Tcl supported the syntax, the following two calls would be equivalent:

```
utlGetArray x $i $j
set x($i)($j)
```

Since Tcl doesn't support that syntax, this function uses the less-convenient syntax that Tcl does support. The following two calls are equivalent:

```
[utlGetArray x $i]
$x($i)
```

The *arrayName* parameter is the name of the nested array.

The *index1 ... indexN* parameters indicate which element in the array to extract.

Return Value

Returns the value in the indicated element in *arrayName*. If no such element exists, a Tcl error is raised.

See Also

"utlPrintArray" on page 8-62, "utlSetArray" on page 8-63, and "utlUnsetArray" on page 8-80

utlHelpGui

Creates, configures, and displays the help window.

Synopsis

utlHelpGui [**-height** *h*] [**-width** *w*] *panel message file_name*

Description

utlHelpGui creates, configures, and displays the help window. The help window contains two important areas:

- an output-only message area
- an input and output user-notes area

The output-only message area can display help text. The input and output user-notes area allows end users to write their own notes for later viewing. These notes are automatically stored in user-note files.

The **-height** *h* option sets the height of the message area to *h* rows.

The **-width** *w* option sets the width of the message area to *w* columns. The default width depends upon the length of the help message: longer help messages increase the width of the help window.

The *panel* parameter specifies the name to assign to the help window's top-level window.

The *message* parameter specifies the help message to display.

The *file_name* parameter specifies the name of the user-notes file. The global variable *utl_help_dir* specifies the name of the directory that help files reside in. If *utl_help_dir* is not set the first time that **utlHelpGui** is called, *utl_help_dir* will default to the current working directory.

Caution

If the directory containing the user-notes file could not be auto-created, is not a directory, or does not provide write permissions, the user-notes portion of the help window will be omitted.

Return Value

No return value is defined.

utlIsNull

Determines if *variable* is empty.

Synopsis

utlIsNull *variable*

Description

utlIsNull determines if *variable* is empty. The following variable values are defined to be empty:

```
""
"{}"
```

Return Value

Returns 1 if *variable* is empty and 0 if *variable* is not empty.

utlIsPanel

Determines if *panel* is a valid Tk control panel.

Synopsis **utlIsPanel** *panel*

Description **utlIsPanel** determines if *panel* is a valid Tk control panel. A panel is only considered to be valid if it exists and contains one or more control panel widgets.

Return Value Returns 1 if *panel* is a valid Tk control panel, otherwise returns 0.

utlJoinPathVar

Converts a Tcl list of paths into a single path value that can be assigned to the path environment variable.

Synopsis

utlJoinPathVar *list*

Description

utlJoinPathVar joins component paths specified by the Tcl list *list* for use as the value of a path environment variable. Separator characters are added as appropriate.

utlJoinPathVar is useful in Tcl scripts that must run on both UNIX and Windows NT systems.

Return Value

Returns a character string consisting of paths that are appropriately joined.

See Also

"utlSplitPathVar" on page 8-68

utlList2Path

Converts a Tcl list to a path specification.

Synopsis

utlList2Path [**-separator** *sep*] [**-quote** *q*] *list*

Description

utlList2Path converts a Tcl *list* to a path specification. In the resulting path specification, list elements are separated by a *sep* character. Each *sep* character embedded in a path component is preceded by the quote character *q* in the returned result.

The **-separator** *sep* option sets the separator character to *sep*. The default separator character is a solidus (/).

The **-quote** *q* option sets the quote character to *q*. The default quote character is a reverse solidus (\).

Return Value

Returns the resulting path specification.

See Also

"utlPath2List" on page 8-57.

utlMkPanelPrologue, utlMkPanelEpilogue, utlMkPanelVisible

Constructs windows and makes an existing window visible.

Synopsis

utlMkPanelPrologue [**-title** *name*] [**-class** *class*] *name*

utlMkPanelEpilogue [**-dialog**] [**-assist**] [**-sameSize | -sameWidth | -sameHeight**] *panel*

utlMkPanelVisible [**-state** *state*] [**-noRaise**] [**-noWiggle**] [**-sameWidth**] [**-sameHeight**] *panel*

Description

utlMkPanelPrologue and **utlMkPanelEpilogue** construct windows. **utlMkPanelVisible** makes an existing window visible to the end user.

utlMkPanelPrologue creates a top-level window if necessary, withdraws this window so that the end user is not distracted by window construction activity, assigns the window's title, class and icon, and sets up a window-creation error handler. The window-creation error handler ensures partially created windows are never left lying around. Calls to this utility should eventually be followed by calls to **utlMkPanelEpilogue**.

The **-title** option sets the window's title to *name*.

The **-class** option sets the window's class to *class*.

utlMkPanelEpilogue destroys the window-creation error handler set up by **utlMkPanelPrologue**, positions the window on the display, makes the window resizable, maps the window, and wiggles it if configured to do so. Calls to this utility should be preceded by calls to **utlMkPanelPrologue**.

The **-dialog** option changes the window to a dialog window that the end user will have to deal with before dealing with any other.

The **-assist** option places the window beside the cursor rather than according to the current window positioning policy. This is convenient, for example, for completion windows.

The **-sameSize**, **-sameWidth**, or **-sameHeight** option sets the resulting window to the same size, width, or height (respectively) as it had the last time it was visible.

utlMkPanelVisible maps the window if it is unmapped, deiconifies the window if it is iconified, raises the window, and optionally wiggles the window to attract the end user's attention.

The **-noRaise** option disables the automatic raising of the window.

The **-noWiggle** option disables the window wiggle.

The **-state** option, where *state* is either *iconic* or *normal*, leaves the window in the iconified or deiconified state, respectively.

Normally **utlMkPanelVisible** updates the window's minimum width and height, but if either the **-sameWidth** or **-sameHeight** option is supplied, the window's minimum width and height remains unchanged.

Return Value

No return value is defined.

Example

```
# Usage: myProcGui [-foo] panel
proc myProcGui args {
   utlBusyCursor

   #  fetch optional parameter
   set f [utlGetArg "-foo" ""]

   #  fetch required parameter
   set p [utlGetArg]
   utlArgEnd

   if {[utlIsPanel $p]}{
      < update panel widgets if/as necessary here >
      utlMkPanelVisible $self
      utlIdleCursor
      return
   }

   #  prepare top-level window
   utlMkPanelPrologue p -title [utlNls "My Title"] \
      -class "myClassName"

   #  create frame widgets
   frame ${p}myFrame1_fr
   frame ${p}myFrame2_fr
     :         :
   utlPrepareWidget ${p}myFrame1_fr ${p}myFrame1_fr ...

   #  create window widgets
   < create and configure Tk widgets here >

   #  pack widgets together
   < pack Tk widgets here >

   #  publish window
   utlMkPanelEpilogue $self

   utlIdleCursor

}
```

See Also

"utlPrepareWidget" on page 8-60 and "utlFocusTraversal" on page 8-39.

utlNls

Returns a localized string.

Synopsis

utlNls *key arg0 arg1 ...*

Description

utlNls returns a localized string for the *key* parameter. The global array *utl_msg_cat* contains the localized messages. The *key* parameter is an index into this array and is also the default message string. Therefore, if *utl_msg_cat($key)* does not exist, *key* is the string returned.

utlNls supports the ability to insert substrings into the localized string. This is done by passing extra arguments to **utlNls**, then referencing these arguments in the localized string with *%0*, *%1*, and so forth. *%0* is replaced by the contents of *arg0*, *%1* is replaced by the contents of *arg1*, and so forth.

Return Value

The localized and formatted string.

The examples below assume that the message catalog file contains the following four definitions:

Example

```
set utl_msg_cat("Password") "Password"
set utl_msg_cat(my_long_message) "This is a long message .... "
set utl_msg_cat(file_not_found) "The file %0 was not found."
set utl_msg_cat("Copy %0 to %1?") "Create the clone %1 from %0?"
```

The following examples return various message strings and assign them to Tcl variables.

Example

```
# assign to "n" the default string
set n [utlNls "User Name"]

# assign to "p" a localized short string
set p [utlNls "Password"]

# assign to "l" a long string
set l [utlNls "my_long_message"]

# assign to "f" a formatted long string
set file_name "my.file"
set f [utlNls file_not_found $file_name]

# assign to "c" a formatted short string
set src "mysource.file"
set dest "mydest.file"
set c [utlNls "Copy %0 to %1?" $src $dest]
```

utlOpen, utlClose

Opens and closes files.

Synopsis

utlOpen *name mode*

utlClose *f*

Description

utlOpen opens files and **utlClose** closes files. These utilities are similar to Tcl's **open** and **close** commands, but provide support for file versioning and for safe file writes.

With file versioning, if the file *foo* is opened for write and a file named *foo* already exists, the existing file *foo* is renamed *foo,v2* and the newly opened file is named *foo*. If a file named *foo,v2* also exists, this file is renamed *foo,v3* before the existing file *foo* is renamed *foo,v2*. In general, files named *foo,v(n)* are renamed to *foo,v(n+1)* as long as *n* is less than *utl_max_file_version*. For files whose version number is greater than or equal to *utl_max_file_version*, the files are overwritten.

With safe writes, when the file *foo* is opened for write, the writes are actually written to a file named *foo.tmp*. When the file is eventually closed, the *foo.tmp* file is renamed *foo*. Safe writes ensure that if the file *foo* exists and that it is 100% complete.

Return Value

utlOpen returns an open file descriptor. No return value for **utlClose** is defined.

If any of the conversion characters (y, m, d, H, M, S, or U) are to appear in *format* as literal characters, they must be quoted by preceding each of them with a reverse-solidus character (\).

The *string* parameter specifies the string to be parsed.

If the **-wild** option is supplied, any field in *string* may be a positive integer, a negative integer, or the asterisk character (*). Otherwise, all fields in *string* must be positive integers except, of course, for the character-oriented fields: *mmm*, *mmmm*, *ddd*, *dddd*, *AM/PM*, and *am/pm*.

Return Value

Returns a list containing the fields (*year month day hour minute second usec*). Unspecified fields are set to the empty list {}.

For the returned fields:

```
    year == the year: 1970-2069 or *
   month == the month: 1-12 or *
     day == the day of the month: 1-31 or *
    hour == the hour of the day: 0-23 or *
  minute == the minute of the hour: 0-59 or *
  second == the second of the minute: 0-60 or *
    usec == the number of microseconds: 0-1000000 or *
```

See Also

"utlFormatTime" on page 8-40.

utlPath2List

Converts a path specification to a Tcl list.

Synopsis

utlPath2List [**-separator** *sep*] [**-quote** *q*] *path*

Description

utlPath2List converts a path specification to a Tcl list. In the path specification, list elements are separated by a *sep* character, except for *sep* characters preceded by the quote character *q*.

The **-separator** *sep* option sets the separator character to *sep*. The default separator character is a solidus (/).

The **-quote** *q* option sets the quote character to *q*. The default quote character is a reverse solidus (\).

Return Value

Returns the resulting Tcl list specification.

See Also

"utlList2Path" on page 8-51.

utlPathVarSeparator

Returns the character used to separate the components of a path environment variable.

Synopsis

utlPathVarSeparator

Description

utlPathVarSeparator fetches and uses the appropriate character when separating the components of a path environment variable. On UNIX systems this character is a colon (:), while on Windows NT systems this character is a semi-colon (;).

utlPathVarSeparator is useful in Tcl scripts that must run on both UNIX and Windows NT systems.

Return Value

Returns the appropriate path separator character.

utlPattern2RegExp

Converts a pattern into a regular expression.

Synopsis **utlPattern2RegExp** *pattern*

Description Converts the specified pattern into a regular expression.

Return Value The regular expression corresponding to the pattern.

utlPrepareWidget

Prepares the specified widgets for display.

Synopsis

utlPrepareWidget [**-autoSelect** *list*] [**-errScheme**] [**-multiListBox**] [***-noEntry***] [**-font** *name* | *dynamic* | *dynamic-bold* | *computer* | *static* | *static-bold*] *widget1 widget2 widget3 ...*

Description

utlPrepareWidget prepares the specified widgets for display. All newly created Tk widgets should be processed by this utility. Each widget only needs to be processed once.

For entry and text widgets that are output only, use the **-noEntry** option to ensure the widget's background color is appropriately set. The background color for widgets that accept textual input is different from the usual background color. This allows the end user to more easily identify data entry areas.

The ***-font*** *name* option sets the font to *name*. *name* can either be the actual name of a font or a class name such as *dynamic*, *dynamic-bold*, *computer*, *static*, or *static-bold*. You should use nonproportional fonts for text entry widgets (dynamic) and proportional fonts for output only text widgets (static).

The **-multiListBox** option enables multiple selections in listbox widgets. By default, listbox widgets only support single selections.

When a checkbutton or radiobutton widget controls whether or not an entry widget is enabled, use the **-autoSelect** option when preparing the entry widget. This option expects one argument: a list of all checkbutton and radiobutton widgets that affect the state of the entry widget. This option arranges for the listed checkbutton and radiobutton widgets to automatically become enabled when the end user clicks a mouse button on the entry widget.

utlPrepareWidget assigns widgets their colors. There are two color schemes for windows:

- The normal color scheme.
- The error color scheme.

By default **utlPrepareWidget** assigns widgets the normal color scheme, but if the **-errScheme** option is supplied, the error color scheme is used.

utlPrepareWidget also sets the frame width of the specified widget.

Return Value

No return value is defined.

See Also

"utlFocusTraversal" on page 8-39 and "utlMkPanelPrologue, utlMkPanelEpilogue, utlMkPanelVisible" on page 8-52.

utlPrintArray

Prints array variables to an open file descriptor. The arrays may be normal Tcl arrays or nested arrays.

Synopsis

utlPrintArray *fd arrayName pattern*

Description

Prints the contents of the array specified by *arrayName* to the open file descriptor specified by *fd*. If the optional *pattern* parameter is supplied, only array indices matching this pattern are printed. *pattern* must be expressed using pattern matching notation.

Return Value

No return value is defined.

See Also

"utlGetArray" on page 8-46, "utlSetArray" on page 8-63, and "utlUnsetArray" on page 8-80.

utlSetArray

Sets a value in a nested array.

Synopsis

utlSetArray *arrayName index1 ... indexN value*

Description

utlSetArray sets a value in a nested array. If Tcl supported the syntax, the following two calls would be equivalent:

```
utlSetArray x $i $j 3
set x($i)($j) 3
```

Since Tcl doesn't support that syntax, this function uses the less-convenient syntax Tcl does support. The following two calls are equivalent:

```
utlSetArray x $i 3
set x($i) 3
```

The *arrayName* parameter is the name of the nested array.

The *index1 ... indexN* parameters are the indicated element in the array.

The *value* parameter is the value assigned to the indicated element in the array.

Return Value

Always returns *value*.

See Also

"utlGetArray" on page 8-46, "utlPrintArray" on page 8-62, and "utlUnsetArray" on page 8-80

utlSharedLibSuffix

Returns the shared-library suffix used on the host platform.

Synopsis **utlSharedLibSuffix**

Description **utlSharedLibSuffix** determines the suffix used by shared-library file names on the host platform.

Return Value Returns the suffix for shared-library files.

utlSharedLibVarName

Returns the name of the environment variable used for shared library search paths on the host platform.

Synopsis **utlSharedLibVarName**

Description **utlSharedLibVarName** determines the name of the shared-library search path environment variable for the host platform.

Return Value Returns an environment variable name.

utlShiftTimeZone

Converts a date/time string from one time zone to another.

Synopsis

utlShiftTimeZone [-**from** *zone*] [-**to** *zone*] *format string*

Description

utlShiftTimeZone converts a date/time string from one time zone to another.

The -**from** *zone* and -**to** *zone* options allows you to specify a time zone for the returned date/time string. *zone* can be assigned two different types of values: a time zone name, or a full time zone specification. For a time zone name, see "utlCurrentTime" on page 8-30 for some possible values supported on HP-UX systems.

For a full time-zone specification, *zone* must have the following form:

```
[:]STDoffset[DST[offset][,rule]]
```

where

STD and DST are one or more bytes that designate the standard time zone (STD) and summer, or daylight savings time zone (DST). STD is required. If DST is not specified, summer time does not apply in this locale. Any characters other than the numerals 0 through 9, the comma (,) character, the minus (-) character, or the plus (+) character are allowed.

offset is the value that must be added to local time to arrive at Coordinated Universal Time (UTC). offset has the following form:

```
hh[:mm[:ss]]
```

where

the hour (hh) is any value from 0 through 23. The optional minutes (mm) and seconds (ss) fields are a value from 0 through 59. The hour field is required.

If offset is preceded by a minus (-) character, the time zone is east of the Prime Meridian. If offset is preceded by a plus (+) character, the time zone is west of the Prime Meridian. The default case is west of the Prime Meridian.

rule indicates when to change to and from daylight savings time. rule has the following form:

```
date/time,date/time
```

where

the first date/time specifies when to change from standard to daylight savings time, and the second date/time specifies when to change back. These fields are expressed in current local time. The form of date should be one of the following:

Dm.d Day of the month, where m is the month (1 through 12) and d is the day of the month (1 through 31).

Jn Julian day (1 through 365). Does not count leap days (February 29).

n The zero-based Julian day (0 through 365). Counts leap days (February 29).

Mm.n.d The day of the week of the month, where m is the month (1 through 12), n is the week of the month (1 through 5, with 1 being the week in which the first day of the month falls) and d is the day of the week (0 through 6, with 0 being Sunday).

time has the same format as offset except that no leading sign ("-" or "+") is allowed. The default, if time is not given, is 02:00:00.

If the -**from** option is not specified, *string* is assumed to be in the local time zone. If the -**to** option is not specified, the returned date/time string will be expressed in local time.

format specifies the required format for the returned date/time character string. **utlCurrentTime** supports the same date/time formats as the utility function **utlFormatTime**.

string specifies which date/time string to convert.

Return Value

Returns a formatted date/time string in the same format as *string*.

utlSplitPathVar

Converts the value of a path environment variable into a Tcl list of paths.

Synopsis

utlSplitPathVar *string*

Description

utlSplitPathVar splits the path variable specified by *string* into its component paths.

utlSplitPathVar is useful in Tcl scripts that must run on both UNIX and Windows NT systems.

Return Value

Returns the component paths as a Tcl list.

utlStrAlign

Pads the supplied strings with space characters.

Synopsis

utlStrAlign [**-left**] *string1 string2 ...*

Description

utlStrAlign pads the supplied strings with space characters so that they all have the same number of characters (in other words, are the same length). The default is to align the right edge by appending space characters to each string.

If the **-left** option is supplied, the strings will be aligned against the left edge by appending space characters to each string.

Return Value

Returns a Tcl list of aligned strings.

Example

```
set sa [utlStrAlign -left "Name:" "Password:"]
set aligned_name   [lindex $sa 0]
set aligned_passwd [lindex $sa 1]
```

utlStrToFname

Converts a string to a file name.

Synopsis

utlStrToFname *string*

Description

utlStrToFname converts a string to a file name. This utility encodes characters that are not allowed or are awkward in file names, such as white space, solidus, and brace characters.

Return Value

Returns the resulting file name.

See Also

"utlFnameToStr" on page 8-38.

utlTableHeader, utlTableRow, utlTablePut

Composes and prints tabular data.

Synopsis

utlTableHeader [**-title** *title*] *f heading1 heading2 ...*

utlTableRow *f column1 column2 ...*

utlTablePut [**-indent** *n*] [**-justify** *left* | *right* | *center*] [**-widths** *w*] *f*

Description

utlTableHeader, **utlTableRow**, and **utlTablePut** compose and print tabular data. Tables are first composed and then printed. You may simultaneously construct more than one table at a time, but you may only construct one table at a time for each file descriptor.

utlTableHeader initiates table construction, and optionally assign the table a title and column headings. Tables may have one or more columns. If no headings are specified, the table will have one column (the minimum).

utlTableRow adds one row of data to the table associated with the open file descriptor *f*. If no *columnN* parameters are supplied, a blank separator row is added to the table. Calls to **utlTableRow** must be preceded by a call to **utlTableHeader**.

utlTablePut writes the previously composed table that is associated with the open file descriptor *f* to the file/stream associated with *f*. Calls to **utlTablePut** must be preceded by a call to **utlTableHeader**, and zero or more calls to **utlTableRow**.

The **utlTablePut** option **-indent** *n* indents the generated table by *n* columns. The default indentation is 0 columns.

The **utlTablePut** option **-justify** *j* allows you to specify how to justify the text in each column. *j* is a Tcl list with one item for each column in the table. The value for each item may be left, right, or center. The default justification for all columns is left.

The **utlTablePut** option **-widths** *w* allows you to specify the width of each column in characters. *w* is a Tcl list with one item (a positive integer) for each column in the table. The default column width is the number of columns required by the widest item appearing in the column.

Return Value

No return value is defined.

Example

```
set f [open "myfile.out" w]

utlTableHeader $f -title "My Table" "Column 1" "Column 2"

utlTableRow $f "Product" "my product"
utlTableRow $f "Vendor" "Hewlett-Packard"
utlTableRow $f
utlTableRow $f "Version" "my version"
utlTableRow $f "Version Date" "my date"

utlTablePut $f -indent 2 -justify {right left}

close $f
```

utlTimerNextTimeout

Calculates the next timeout date.

Synopsis

utlTimerNextTimeout [-**zone** *zone*] [-**from** *time*] *timeoutSpec*

Description

The **utlTimerNextTimeout** utility calculates the date and time of the next timeout for *timeoutSpec*.

The -**zone** *zone* option allows you to specify a time zone for the returned date/time string. *zone* can be assigned two different types of values: a time zone name, or a full time zone specification. For some possible time zone name values, see "utlShiftTimeZone" on page 8-66.

The -**from** *time* option allows you to establish the base time as either the number of seconds since the epoch, or a date and time string in the form of a six- or seven-item Tcl list:

```
{year month day hour minute second}
{year month day hour minute second usecs}
```

The *timeoutSpec* parameter is the timeout specification as either the number of seconds until the next timeout event, the letter *Q* followed by the number of seconds until the next timeout event, or a date and time in the form of a six- or seven-item Tcl list:

```
{year month day hour minute second}
{year month day hour minute second usecs}
```

where

year is the year, typically any value between 1970 and 2032.

month is the month, typically any value between 1 and 12.

day is the day of the month, typically any value between 1 and 31.

hour is the hour, typically any value between 0 and 23.

minute is the minute, typically any value between 0 and 59.

second is the second, typically any value between 0 and 59.

usec is microseconds, typically any value between 0 and 999999.

Alternatively, each of the above items can have the following form:

Format	Description
empty	A wild card, accept any valid value.
num	Use the supplied *num*.
+*num*	Use the sum of the current time and *num*.
-*num*	Use the next larger current time component divisible by the supplied *num*.

In addition, the *year* field can be set to the following strings:

String	Description
leap-year	Make the year a leap year.
non-leap-year	Make the year a non-leap year.

In addition, the *month* field can be set to the following strings:

String	Description
Easter-Sunday	Use the month in which Easter Sunday falls.
Easter-Monday	Use the month in which Easter Monday falls.
Good-Friday	Use the month in which Good Friday falls.
Boxing-Day	Use the month in which Boxing Day falls.

In addition, the day field can be set to the following strings:

String	Description
Easter-Sunday	Use the day on which Easter Sunday falls.
Easter-Monday	Use the day on which Easter Monday falls.
Good-Friday	Use the day on which Good Friday falls.
Boxing-Day	Use the day on which Boxing Day falls.
Statutory-Holiday	Use days that fall on a statutory holiday.
non-Statutory-Holiday	Use days that do not fall on a statutory holiday
Holiday	Use days that fall on statutory holidays and on weekends.

String	Description
non-Holiday	Use days that fall on business days.
weekend	Use days that fall on a weekend.
weekday	Use days that don't fall on a weekend.
last	Use the last day of the month.
week—*day_name*	Use a specified day of week where *week* is the 1st, 2nd, 3rd, 4th, 5th, or last. *day_name* is Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday.
day_name < = *day*	Indicates use the day of the week on or before *day*.
day_name > = *day*	Indicates use the day of the week on or after *day*.
day_name—closest—*day*	Indicates use the day of the week closest to *day*.

Return Value

Returns the date and time of the next timeout as a seven-item Tcl list:

```
{year month day hour minute second usecs}
```

See Also

"utlTimerQuery" on page 8-76, "utlTimerStart" on page 8-77, and "utlTimerStop" on page 8-79

utlTimerQuery

Retrieves timer information.

Synopsis

utlTimerQuery [**-id** *id*] [*what1* [*what2* [*what3* [...]]]]

Description

utlTimerQuery retrieves timer information.

The **-id** option queries the timer specified by *id*, which is the value returned by **utlTimerStart**.

whatX specifies the object of the query. Some possible values for *what* are as follows:

command timeout command or commands

isRunning is a specific timer or any timer running?

tclID Tcl timer ID or IDs

timers IDs for running timer or timers

Return Value

Returns the results of the query.

See Also

"utlTimerNextTimeout" on page 8-73, "utlTimerStart" on page 8-77, and "utlTimerStop" on page 8-79

utlTimerStart

Starts a one-shot or periodic timer.

Synopsis

utlTimerStart [-**zone** *zone*] [-**repeat** *time*] [-**id** *id*] *start_time command*

Description

The -**zone** option sets the time zone for the passed-in dates and times, as specified by *zone*. See "utlCurrentTime" on page 8-30 for possible values.

The -**repeat** option sets the repeat criteria of a periodic timer. *time* specifies either the number of seconds between timeout events, the letter *Q* followed by the number of seconds between timeout events, or a date and time in the form of a six- or seven-item Tcl list (see the *start_time* parameter below). If the -**repeat** option is not supplied, a one-shot timer starts.

The -**id** option passes the timer ID specified by *id* to the timer. *id* must not match the ID of a timer that is already running.

The *start_time* parameter indicates when to start the timer. This can be specified as either an empty list—which tells the timer to start immediately—the number of seconds since the epoch, or the date and time in the form of a six- or seven-item Tcl list:

```
{year month day hour minute second}
{year month day hour minute second usecs}
```

where

year is the year, typically any value between 1970 and 2032.

month is the month, typically any value between 1 and 12.

day is the day of the month, typically any value between 1 and 31.

hour is the hour, typically any value between 0 and 23.

minute is the minute, typically any value between 0 and 59.

second is the second, typically any value between 0 and 59.

usec is microseconds, typically any value between 0 and 999999.

See "utlTimerNextTimeout" on page 8-73 for alternative values for *year*, *month*, *day*, *hour*, *minute*, *second* and *usecs*.

The *command* parameter indicates which Tcl command to execute upon timeout. The following character strings are replaced with the appropriate values before *$command* is executed:

- *%time*" Replaced with the current time in seconds as floating-point number.
- %datetime" Replaced with The current date and time as a Tcl list:

  {*year month day hour minute second usecs*}
- *%id*" Replaced with the timer's timer ID.
- *%delay*" Replaced with the time in seconds until the next timeout as a floating-point number.
- *%zone*" Replaced with the time zone name.

Return Value

Returns a timer ID.

See Also

"utlTimerNextTimeout" on page 8-73, "utlTimerQuery" on page 8-76, and "utlTimerStop" on page 8-79

utlTimerStop

Stops the specified timer.

Synopsis **utlTimerStop** *id*

Description Stops the timer specified by the parameter *id*.

Note Stopping a timer that is no longer running will not produce an error.

Return Value No return value is defined.

See Also "utlTimerNextTimeout" on page 8-73, "utlTimerQuery" on page 8-76, and "utlTimerStart" on page 8-77

utlUnsetArray

Deletes array variables, including nested arrays.

Synopsis **utlUnsetArray** *arrayName*

Description **utlUnsetArray** does the same thing as the Tcl command *unset arrayName*, but works around memory leaks in [incr Tcl] version 1.5.

Return Value No return value is defined.

See Also "utlGetArray" on page 8-46, "utlPrintArray" on page 8-62, and "utlSetArray" on page 8-63.

utlWidgetState

Enables and disables Tk widgets.

Synopsis

utlWidgetState [**-normal**] [**-disabled**] [**-errScheme**] *widget1 ... widget*

Description

utlWidgetState enables or disables the widgets specified by *widgetN*. The **-normal** flag enables the widgets, while the **-disabled** flag disables the widgets. If the **-errScheme** flag is supplied, the error color scheme is used rather than the normal color scheme.

The **utlWidgetState** is not needed for Tk widgets that support the **-state** configuration option—only for those widgets that don't support this option (such as scrollbar and label widgets).

For those Tk widgets supporting the **-state** configuration option, **utlWidgetState** changes the widget's state using the Tk *configure* command. For all other widgets, **utlWidgetState** modifies the color of the widget to make it appear disabled or enabled. For example, the text of the Tk label widget appears grayed-out when the widget is disabled, and normal when the widget is enabled.

Return Value

No return value is defined.

utlWidgetText

Returns the user-visible text that identifies the specified widget.

Synopsis

utlWidgetText *widget*

Description

utlWidgetText returns user-visible text that identifies (to the end user) the specified widget on a window. Any leading or trailing white space is deleted from the result before it is returned.

Return Value

Returns the user-visible text associated with *widget*.

Caution

Some widgets, such as frames, don't contain any identifying text.

See Also

"utlChkAlpha" on page 8-20, "utlChkInt" on page 8-23, and "utlChkName" on page 8-24.

yourIntfCompletionList

Composes a list of completions.

Synopsis

yourIntfCompletionList *name*

Description

yourIntfCompletionList composes a list of completions for the type of target system specified by the prefix name **yourIntf**. This utility traverses the target system's name-space hierarchy to compose the list.

The *name* parameter specifies the initial path name, as a Tcl list, upon which to base the returned completions.

Return Value

Returns the possible completions as a Tcl list.

Index

About this Edition

March 2000: Fifth Edition. Revised to describe Enterprise Link Edition E.02.30 for Windows NT 4.0. For details about what is new in this release, refer to the Enterprise Link E.02.30 Release Notes.

November 1999: Fourth Edition. Revised to describe Enterprise Link Edition E.02.20 for HP-UX 10.20 and Windows NT 4.0. (Documentation released in only PDF and HTML versions.)

December 1997: Third Edition. Revised to describe Enterprise Link version E.02.00.

April 1997: Second Edition. Revised to describe Enterprise Link 1.10 for HP-UX 10.20 and for Windows NT 4.0.

July 1996: First Edition.

Need Assistance?

Customer Support

If you need technical assistance, refer to your Enterprise Link support contract. Your Enterprise Link support contract provides the telephone number for contacting the appropriate Agilent Tecnologies customer response center.

Enterprise Link support is provided as per your support contract. See your Enterprise Link support contract for details.

When contacting Agilent Technolgies, please have the following information available:

- your name
- your company's name
- your phone and fax numbers
- your system's configuration, including:
  - the version of Enterprise Link you are using and any options that are in use
  - the name and version number of your operating system
  - the manufacturer and model of your computer
  - your system's hard disk size
  - the amount of memory installed on your system
- errors reported by the system
- a description of the problem, including the steps that lead to the problem

Enterprise Link on the Internet

Visit the Automation Integration Software web site for more Enterprise Link information. Visit the web site at **http://www.agilent.com/find/ais**