Bellcore
Bell Communications Research

BELLCORE PRACTICE
BR 007-252-008
ISSUE 3, OCTOBER 1997
RELEASE 5.2

MYNAH™ System Scripting Guide

Bellcore
Bell Communications Research

BELLCORE PRACTICE
BR 007-252-008
ISSUE 3, OCTOBER 1997

MYNAH™ System Scripting Guide

For further information, please contact:

MYNAH Customer Service Center

1-(732) 699-2668, option 3

To obtain copies of this document, call (732) 699-5802.

Bellcore
Bell Communications Research

Document Feedback

We at Bellcore are constantly striving to meet your need for information. Once you've had a chance to use this document that we've written for you, please let us know if it met your needs. Please complete this form and either FAX it to us at (908) 336-2995 or return it to us at the address below.

Document No.	Issue No.	Publication Date	Revision No.	Supplement No.
BR 007-252-008	Issue 3	October 1997		

1. In each of the following areas, how well did this document meet your need for information?

	Missed	Nearly Met	Met	Exceeded	Not Applicable
a. Relevance of the information to your work	☐	☐	☐	☐	○
b. Ease of finding the information that you need	☐	☐	☐	☐	○
c. Clarity of the information	☐	☐	☐	☐	○
d. Accuracy of the information	☐	☐	☐	☐	○
e. Usefulness of the information	☐	☐	☐	☐	○
f. Thoroughness of the information	☐	☐	☐	☐	○
g. Level of detail of the information	☐	☐	☐	☐	○
h. Availability of this document when you needed it	☐	☐	☐	☐	○
i. Overall quality of this document	☐	☐	☐	☐	○

2. Please comment on any of the areas where this document *did not* meet or exceed your need for information.

_____

_____

3. Are there features of this document that you found particularly useful or informative? Please explain.

_____

_____

4. Are there other ways that we can improve this document? Please feel free to comment on any aspect of it.

_____

_____

5. For what purpose did you use this document?

☐ As a technical reference ☐ To use a system ☐ To learn methods/procedures

☐ As an administrative reference ☐ To install/administer a system ☐ To be better informed

☐ Other (please specify) _____

6. Please tell us something about yourself.

Your company/employer _____ Your title _____

Your job responsibilities _____

If you would like us to let you know what we're doing in response to your feedback, please write your name and address (or telephone number) below.

Name _____ Telephone Number _____

Address _____

Thank you for your time and cooperation!

To return this form, please FAX it to (908) 336-2995, or mail it to Ken Berczik, Bellcore Learning Support, 444 Hoes Lane, Room RRC 4F-808, Piscataway, NJ 08854.

NOTICE OF DISCLAIMER AND LIMITATION OF LIABILITY

This document is intended for use solely by Bellcore customers who have licensed the Bellcore software described herein. The software, this document, and the information contained within this document may be used, copied or communicated only in accordance with the terms of a written license agreement with Bellcore. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording without the prior written permission of Bellcore. While the information contained herein has been prepared from sources deemed reliable, Bellcore reserves the right to revise the information without notice, but has no obligation to do so. Unless the recipient has been expressly granted a license by Bellcore under a separate applicable written agreement with Bellcore, no license, express or implied, is granted under any patents, copyrights or other intellectual property rights. Use of the information contained herein is in your sole determination and shall not be deemed an inducement by Bellcore to infringe any existing or later-issued patent, copyright or other intellectual property rights.

BELLCORE PROVIDES THIS DOCUMENT “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR AGAINST INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS. FURTHER, BELLCORE MAKES NO REPRESENTATION OR WARRANTY, EXPRESS OR IMPLIED WITH RESPECT TO THE SUFFICIENCY, ACCURACY, OR UTILITY OF ANY INFORMATION OR OPINION CONTAINED HEREIN. BELLCORE EXPRESSLY ADVISES THE USER THAT ANY USE OF OR RELIANCE UPON SAID INFORMATION OR OPINION IS AT THE SOLE RISK AND LIABILITY, IF ANY, OF THE USER AND THAT BELLCORE SHALL NOT BE LIABLE FOR ANY DAMAGE OR INJURY INCURRED BY ANY PERSON ARISING OUT OF THE SUFFICIENCY, ACCURACY, OR UTILITY OF ANY INFORMATION OR OPINION CONTAINED HEREIN. BELLCORE, ITS OWNERS AND AFFILIATES SHALL NOT BE LIABLE WITH RESPECT TO ANY CLAIM BEYOND THE AMOUNT OF ANY SUM ACTUALLY RECEIVED IN PAYMENT BY BELLCORE FOR THE DOCUMENTATION, AND IN NO EVENT SHALL BELLCORE, ITS OWNERS OR AFFILIATES BE LIABLE FOR LOST PROFITS OR OTHER INCIDENTAL, OR CONSEQUENTIAL DAMAGES.

Bellcore does not recommend or endorse products and nothing contained herein is intended as a recommendation or endorsement of any product.

For further information, please contact:

The MYNAH Customer Service Center 8:00 AM and 7:00 PM ET Monday through Friday, (732) 699-2668, Option . If outside the 732 area, call (800) 795-3119, Option 3.

You can also contact support (for non critical problems) via e-mail at mynah-support@cc.bellcore.com.

Trademark Acknowledgments

MYNAH System Scripting Guide

Contents

List of Figures

List of Tables

Preface

Document Structure

This document contains a section on the basics of the Tcl language, a section on the TclX extensions, and separate sections on each of MYNAH™ Tcl extensions.

Table 1 list the sections in this document with a brief description of each section.

Table 1. MYNAH Language Guide Sections (Sheet 1 of 2)

Section Number	Section Name	Description
Section 1	Introduction	This section contains general information on this guide, including an overview of concepts shared by all of the MYNAH packages.
Section 2	General Scripting	This section discusses the basics behind scripting, including creation, execution, file output, and database output.
Section 3	Scripting Hints	This section contains hints that you may find useful as you create your scripts, including concealing sensitive data, script termination, and debugging.
Section 4	Tcl Basics	This section is a basic introduction to the Tcl language.
Section 5	xmyVar Global Script Variables	This section contains the complete list of the variables in the global **xmyVar** array.
Section 6	General MYNAH Tcl Extensions	This section describes the set of Tcl extensions that are available to a script automatically when it starts executing as well as describing an array of global MYNAH-specific variables that are provided to the script.
Section 7	Child Script Extension Package	This section describes the set of Tcl extensions for the Child Script Package, which is used to start and control script execution from a parent script.
Section 8	TermAsync Extension Package	This section describes the asynchronous Tcl extensions that provide interactions with an asynchronous terminal device.
Section 9	Term3270 Extension Package	This section describes the 3270 Tcl extensions that provide the functions necessary for interactions with a 3270 device.

Table 1. MYNAH Language Guide Sections (Sheet 2 of 2)

Section Number	Section Name	Description
Section 10	General Application-to-Application Tcl Language Extensions	This section describes the General Application-to-Application (AppApp) extensions that provide the functionality necessary for interaction with a SUT thru an application specific interface.
Section 11	TOP Tcl Language Extension	This section describes the Tcl extensions that provide support for automated interactions with the System Under Test (SUT) on the TOP/TCP/IP application-to-application interface.
Section 12	PRT3270 Tcl Language Extensions	This section describes the Tcl extensions that emulate a 3270 printer so that you can capture messages that a SUT sends to a printer.
Section 13	FCIF Tcl Language Extensions	This section describes the Tcl extensions that process and analyze Flexible Computer Interface Format (FCIF) messages.
Section 14	Message Response Directory Tcl Language Extensions	This section describes the Tcl extensions that provide a mechanism to easily scan all messages that have arrived on a particular communications channel and have been saved to disk.
Section 15	Batch Tcl Language Extensions	This section describes the Tcl extensions that emulate running batch jobs.
Section 16	DCE Extension Package	This section describes the Tcl extensions that provide the functions necessary for interactions with a Distributed Computing Environment (DCE).
Section 17	GUI Tcl Language Extensions	This section describes the Tcl extensions that provide access to GUI test applications.
Appendix A	Basic Tcl Commands	This appendix contains a reproduction of the Tcl manual pages, covering all of the commands in the basic Tcl command set.
Appendix B	Extended Tcl (TclX) Extensions	This appendix describes all of the extensions that are added to Tcl 7.0 by Extended Tcl (TclX 7.0a).
Glossary	Glossary	This section contains a glossary of terms related to the MYNAH System.

NOTE — While separate packages, the combination of the TOPCOM, PRT3270, FCIF, and General App-to-App packages provide complete functions needed to emulate interactions between applications. Therefore, these packages are often referred to as being part of the App-to-App package.

Related Documents

- *BR 007-252-005, MYNAH System Administration Guide*
- *BR 007-252-006, MYNAH System Users Guide*

On-line Versions of the MYNAH Documents

The MYNAH documents are available on-line in the Adobe® Acrobat® PDF format (Release 3.0). The PDF files are accessible from either the local file system or from an internal URL. See the MYNAH administrator for the location of the PDF files.

Viewing the PDF files requires that you have installed the Adobe Acrobat Reader® (Release 3.0). If you need the Acrobat Reader, contact the MYNAH administrator. In addition, you can download the Acrobat Reader directly (off the Internet) from Adobe at *www.adobe.com*.

Once you have installed the Acrobat Reader, you can read the files

- Using the Acrobat Reader directly if the MYNAH System has been installed on a local system.
- Using the Acrobat Reader as plug-in to a browser if the MYNAH System has not been installed on a local system, such as if the system has been installed on a UNIX Solaris™ server and you are using an X-windows™ emulator to access the system on a PC. Consult your browser's documentation for information on how to install plug-ins.

If you access the PDF files via a browser, you may wish to download the files to your local system, which will give you direct access to a file the next time you need to read it, rather than waiting for the browser to load it.

Formatting Conventions

When a term is being defined, e.g. **method**, it appears in Bold Helvetica.

The extension entries are described using the following structure and formatting conventions:

Syntax

```
class_command_name connection_method \
                  -argument_name argument_value\
                  ?-argument_name argument_value?

handle method_name -argument_name argument_value\
                  ?-argument_name argument_value?

handle -attribute_name ?attribute_value?

class_command_name -attribute ?new_default_attribute_value?
```

Return

This is a brief statement of the return value this extension generates.

Description

The syntax section uses the following formatting conventions:

- Class command, method, attribute, and option names appear in `Courier`.
- All substitutable or user supplied items(e.g., handles and argument values) appear in *`Courier Italics`*.
- All optional entries are delimited by question marks.

The description section explains the reasons and uses of the method/attribute with the following conventions:

- Literal strings (class command, method, attribute, and option names) appear in **bold Times**.
- Substitutable or user supplied items (e.g., argument values) appear in ***boldItalics Times***.
- File and directory names appear in *italics Times*.

Exception

Where feasible, each description will contain a section stating the conditions where the extension will throw an exception (fail).

Examples

The examples use the following conventions:

- All examples appear in `Courier`
- The MYNAH System includes a program called **xmytclsh** Section 4.10, that lets you interactively run Tcl commands, including all MYNAH extensions.

When you start the program, the **xmytclsh** prompt (>) appears.

The examples in this document have been created as if they were entered using **xmytclsh**, and follow the following conventions:

- Lines representing text you type begin with the **xmytclsh** prompt (>), e.g.,

  ```
  > xmyTermAsync connect
  ```

- Lines representing return values in examples will also appear in `Courier` but will not begin with the **xmytclsh** prompt, e.g.,

  ```
  > xmyTermAsync connect
  .xmyTermAsync_1
  > set x 5
  5
  ```

  where the line beginning with **5** shows the return value for the ***set*** command.

1. Introduction

This document describes the MYNAH scripting language. This scripting language is based on the Tcl language. Using this language, you will be able to create scripts for executing tests for the various MYNAH packages.

Tcl, pronounced “tickle,” stands for “tool command language” and was developed by John Ousterhout at the University of California at Berkeley as a simple interpretive programming language, implemented as a library of C procedures.

The “flavor” of Tcl used by the MYNAH System is **extended Tcl**, also called **TclX**, under license from NeoSoft. TclX is a superset of standard Tcl and is built alongside the standard Tcl sources, adding extensions to expand Tcl’s capabilities. TclX contains a

- Superset of new commands
- Library of user-extensible Tcl procedures.

Section 4 provides a basic overview of the Tcl language. This section is not meant as an in-depth discussion of the language but as a general introduction. There are, however, several sources for a more complete understanding of Tcl. In addition, there are several Internet-based sources of Tcl information, including the USENET newsgroup *comp.lang.tcl* and several World Wide Web (WWW) home pages. Table 1-1 lists a few of these pages; several of these pages also contain links to other pages on the Web. Table 1-1 also provides the address for NeoSoft’s TclX man pages.

Table 1-1. WWW Tcl Home Pages

Home Page Name	Address
Tcl/Tk Project At Sun Microsystems Laboratories[a]	http://sunscript.sun.com/
A brief introduction to TCL/TK	http://http2.brunel.ac.uk:8080/~csstddm/TCL2/TCL2.html
TCL WWW Info	http://www.sco.com/Technology/tcl/Tcl.html
TclX (TCL) (Neosoft)	http://www.neosoft.com/tcl/TclX.html

a. You can also access this page from John Ousterhout’s home page, http://www.sunlabs.com:80/people/john.ousterhout/.

1.1 MYNAH Extensions Overview

In addition to the standard set of Tcl commands and procedures, the MYNAH System makes use of the TclX extensions as well as sets of MYNAH specific extensions. **Extensions** are commands and procedures that expand Tcl's capabilities. These extensions do not change basic Tcl, they only extend the abilities of the language.

The MYNAH extensions are organized into packages that are related to each other based on the functionality they perform. Each package represents a **domain**, which is the interface between the MYNAH System and the **System Under Test** (SUT). A SUT can be a system you wish to test or can contain the application you wish to automate. For example, the asynchronous terminal interface of an application is a **domain**.

Sections 5 through 17 contain descriptions of the MYNAH extensions to the Tcl language. Each section describes the extension package for a pertinent **domain**.

1.1.1 Extension Types

There are three types of MYNAH extensions:

1. Class Commands - These give you control over a class or category of functions. For more information about Class commands see, Section 1.1.1.1.
2. Methods - These let you perform actions on **instances**, which are connections made to SUTs using a class command. For more information about methods see, Section 1.1.1.4.
3. Attributes - These let you set the configuration characteristics of the instances and class commands. For more information about attributes see, Section 1.1.1.5.

Methods and **attributes** are sub-commands to a **class** command, that is, they cannot be executed independently but rather must be executed in conjunction with a class command.

Table 1-2 contains basic examples of the relationships between class commands, methods, and attributes.

Table 1-2. Example Class Commands, Methods, and Attributes

Package	Class Command	Method	Attribute	Argument	Result
TermAsync	**xmyTermAsync**	**connect**			Creates an asynchronous connection.
TermAsync	**xmyTermAsync**		**-timeout**		Shows the default time-out value for the TermAsync package.
TermAsync	**xmyTermAsync**	**connect**	**-timeout**	***20***	Creates an asynchronous connection with a time-out value of 20 seconds.
General	**xmyExit**			***"exit script now"***	Exits from the current script.

Methods and attributes can be applied independently to class commands. For example, the **connect** method Section 8.4.1.2 can be applied to the **xmyTermAsync** class command Section 8.4 to create an asynchronous connection, e.g.,

```
xmyTermAsync connect
```

So too, the **-timeout** attribute Section 8.4.2.18 can be applied to the **xmyTermAsync** class command to show the default time-out value for the TermAsync package, e.g.,

```
xmyTermAsync -timeout
```

You can use attributes with methods, such as applying the **connect** method and the **-timeout** attribute to the **xmyTermAsync** class command to create an asynchronous connection with a time-out value that is different from the default value, e.g., see the third example in Table 1-2.

```
xmyTermAsync connect -timeout 20
```

You can also apply an argument value directly to a class command. For example, you apply a string that will be produced when exiting from a script directly to the **xmyExit** Section 6.2.7 class command, e.g., see fourth example in Table 1-2.

```
xmyExit "exit script now"
```

The following subsections contain complete explanations of class commands, methods, and attributes.

1.1.1.1 Class Commands

A **class** is a specific area or category of functionality. For example, each MYNAH domain is in essence a class. Thus, the 3270 emulation package (Term3270) is a class. The general MYNAH Package (see Section 6) can also be considered as a class.

A **class command** gives you control over a MYNAH class. For example, the **xmyTerm3270** class command (Section 9.5) gives you control over functions of a 3270 emulation, such as creating a connection. At the same time, **xmyExit** is a general class command used to exit from a script and the Tcl interpreter. Regardless of which type of class command you use, the basic syntax for class commands and their arguments takes one of the following forms:

```
class_command_name method_name ?-attribute attribute_value?

class_command_name -attribute attribute_value ?-attribute
attribute_value?
```

> **NOTE —** Arguments delimited by question marks are optional. See the preface for an explanation of the formatting conventions used in this guide.

The general class commands, those that apply to the entire MYNAH System, are detailed in Section 6. These commands are automatically available to all scripts when they start executing. Among the functions of these class commands are loading MYNAH Tcl extension packages, comparing two files, and changing values in the symbol table.

> **NOTE —** The symbol table is a Tcl list of lists (each sublist containing a variable/value pair) that is passed to scripts at start-up and passed back to the execution user at script termination time. See Sections 6.2.17, 6.2.18, 6.2.19, and 6.2.20 for more information on setting up a symbol table.

The domain-specific class commands are a bit more specialized in that there is only one class command per domain. You use class commands to create and interact with connections to a SUT. In addition, you can also use class commands to set or return default attribute configurations for the entire domain.

Attributes are the characteristics, the pieces of information, of a class or a class's connection, such as the current cursor location or the start-up shell. For example, if you created a connection using the **xmyTerm3270** package (Section 8), such as by executing the following:

```
> set conn2 [xmyTerm3270 connect]
.xmyTerm3270_1
```

you could execute the following command

```
> $conn2 -row
21
```

to find the current row position of the cursor.

See Section 1.1.1.5 for a further discussion on attributes.

One of the main functions of a domain's class command is to create a connection to a SUT. For each domain there is one sub-command (or, in our lexicon, a **method**), called **connect**, that is used by the class command to create a connection. A very basic example would be

```
xmyTerm3270 connect
```

where **xmyTerm3270** is the class command and **connect** is the method.

There is often more to creating a connection than simply typing the name of the class command **xmyTerm3270** and the **connect** method. For example, you may have to specify what you are connecting to. This will be covered in the discussion of each domain's **connect** method. For now, let's just use our basic example of using the **connect** method without any arguments.

1.1.1.2 Instances

When you use the **connect** method, the class command creates a unique connection to a SUT. We call this unique connection an **instance**. For example, every time you invoke **xmyTerm3270 connect**, you create a new instance.

1.1.1.3 Handles

For each instance, the MYNAH System returns a reference to the instance. This reference is called a **handle**. The automatically generated handle name takes the form

```
.class_command_NN
```

where the count number suffix NN is an iterated number assigned by the **Script Engine** (SE), which is the MYNAH process that runs code. For example, the first time you type:

```
> xmyTerm3270 connect
```

the returned handle would be **.xmyTerm3270_1**, the second time it would be **.xmyTerm3270_2**, and so on.

You use the handle to send or receive information from a SUT using the various methods and attributes. For example, the Term3270 Package, used to interact with the 3270 domain,

has a method called **type**, which is used to send keystrokes to a connection. To send the string "tst" to the first handle mentioned above, you would enter:

```
> xmyTerm3270_1 type -text "tst"
```

It is recommended that you assign a handle to a variable. Using the previous **type** method example, you could enter:

```
> set conn1 [xmyTerm3270 connect]
> $conn1 type -text "tst"
```

Without the variable, every time you want to use the handle, you would have to type **.xmyTerm3270_1**. This could get a little tedious. In addition, **.xmyTerm3270_1** may not have much meaning for you, but you could create a value whose name has a specific meaning, such as **$conn3270** or **$connAsync**. Since this is more concise, it will also help clean up your scripts.

Using variables will help shield your scripts from potential connection name changes. For example, if a connection name changes and you used a generated handle name, you would have to change your script, but if you use a variable, the handle will always be set to the variable you use regardless of the format of the system-generated handle name.

The general syntaxes of the handle-based methods and attributes are

handle method_name *arguments*

handle -attribute_name *arguments*

Methods and attributes are explained below.

1.1.1.4 Methods

Methods are sub-commands to class commands. Generally, methods are used to perform actions on instances you create. The **type** method is a good example: it lets you send text to an instance of a 3270 terminal connection.

> **NOTE —** Except for the case described below, do not use methods directly with class commands. For example,
>
> `xmyTerm3270 type -text "tso"`
>
> would be illegal. There is no way of knowing what particular connection you are trying to use **type** to enter this information for; there is no instance to correspond to the method.

We said that methods are generally used to perform particular actions on instances but there is one exception: the **connect** method. **connect** must be used directly with a class command to create an instance, such as

```
set conn1 [xmyTermAsync connect]
```

where the **connect** method is used with the **xmyTermAsync** class command to create a connection to a synchronous SUT.

1.1.1.5 Attributes

The term **attribute** has several meanings in the MYNAH System. At their most basic level, **attributes** are the individual values defining the characteristics, the pieces of information, of a class or a class's connection. For example, attributes can be the name of a host, the cursor's column or row position, the number of seconds the system will wait for a response (i.e., the time-out value).

At the same time, the sub-commands that let you manipulate attribute values are also called **attributes**. Syntactically, attributes can be distinguished from methods in that an attribute name is preceded by a hyphen, as in **-column**.

Attributes are commonly used to set attribute values for an instance or to display the values of those attributes. For example, an attribute of the **$conn1** handle we created in Section 1.1.1.4 is what 3270 terminal model the instance is running as. To display the model (assuming it's a model 2), type

```
> $conn1 -model
2
```

to generate the return value, i.e., 2 for model 2.

Some attributes can only be set upon creation of the connection. Each domain has its own set of attributes that can only be set during connection creation, and this information will be found in each domain's section. Model would be an attribute that can only be set when creating a connection. Other attributes can be set at any time during the life of the instance. For example, the following statements use the **-timeout** attribute to reset the timeout value.

```
> $conn1 -timeout
10

> $conn1 -timeout 30
> $conn1 -timeout
30
```

We first apply the **-timeout** attribute to the **$conn1** handle to see the current timeout value for this instance, which is 10 seconds. We want to change this value to 30 seconds, so we re-apply the **-timeout** attribute, this time adding the value 30 as an argument to the attribute. Lastly, we use the **-timeout** attribute a final time to see that the timeout value has been changed.

Since the **-timeout** attribute was applied to the **$conn1** handle, this change in the timeout value affects this instance only.

Unlike methods, many attributes can also be used directly with class commands to return or set default attribute values, as in

```
> xmyTerm3270 -timeout
10

> xmyTerm3270 -timeout 25
> xmyTerm3270 -timeout
25
```

In this example, we first used **xmyTerm3270 -timeout** to return the current time-out value for the 3270 domain. Then, we used the same class command/attribute combination with an argument to change the default time-out value. All new invocations of **xmyTerm3270 connect** will assume the new default value of 25 seconds.

To summarize, the differences between using attributes with class commands and connection instances are

- This use sets the timeout value for all future Term3270 instances in a script.

  ```
  > xmyTerm3270 -timeout 25
  ```

- This use sets the timeout value for only this instance of a Term3270 connection.

  ```
  > $conn1 -timeout 30
  ```

1.1.2 Extension Functional Categories

The MYNAH Tcl extensions can be organized into six functional categories, as shown in Table 1-3. In the sections describing the MYNAH Package extensions, the methods and attributes are listed alphabetically. At the beginning of each section is a table listing the extensions, organizing them by category.

Table 1-3. MYNAH Tcl Extension Functional Categories (Sheet 1 of 2)

Category Name	Description
Connection	These extensions deal with establishing or destroying a connection or obtaining information about a connection. For example, obtaining a list of open connections, the name of the host, and the connection's status.
Data Entry/Retrieval	These extensions deal with sending or retrieving data from the SUT. You can also obtain information about the data, including screen names and the name of the last pressed function key.
Location	These extensions let you move the cursor or find the location of a screen element.

Table 1-3. MYNAH Tcl Extension Functional Categories (Sheet 2 of 2)

Category Name	Description
Comparisons	These extensions let you compare test data with data returned by a SUT.
Waiting	These extensions give you control over script execution suspension until the SUT has returned with the proper response.
Attributes	In addition to being the individual values defining the characteristics and the sub-commands used to manipulate these values, attributes can also be a category of methods and attributes that are used to find information about the SUT's configuration characteristics, e.g., blinking, highlighted.

2. General Scripting

This section discusses the basic concepts behind scripting, these include the following functionality:

- Creation - Describes the various methods of creating scripts
- Execution - Describes the various methods of executing scripts
- SE States at Start Time - Describes start-up modes
- File Output - Describes the various files that are created when a script is executed
- Database Output - Describes the database objects that are created when a script is executed
- Execution Without Database Update - Describes the steps used to execute a script without updating the database
- Loading Procedures - Involves invoking script commands.

> **NOTE —** See the *MYNAH System Users Guide*, for detailed information on using the **Script Object**, **Script Builder**, **Graphical User Interface** (**GUI**) and the **Command Line User Interface** (**CLUI**) user commands.

2.1 Overview

You may create script code in the MYNAH System to accomplish either of the following:

- Automation of a **test**.
- Automation of a **task.**

This section summarizes how you create and execute scripts.

The remaining sub-sections describe in detail script creation, execution, and execution output.

2.1.1 Creation

Script code is stored in UNIX® files. You can create this code by:

- Using your favorite editor
- Using the MYNAH **Script Object** (code view) window
- Using the MYNAH interactive **Script Builder**.

Script Objects in the MYNAH System database are used to document script code. One of the required attributes of a Script Object is the location of the script code. In other words, a Script Object is an object in the MYNAH database that points to a particular file in the UNIX file system.

Another optional attribute of a Script Object is an association with one or more Test Objects. This attribute is used when the code accomplishes the automation of a Test.

A Script Object is only needed if you want to use the MYNAH database to keep track of your scripts or if you want to track automated testing using a Script Object's database functions.

To make use of all of these database features, one possible scenario might be

1. Create a Script Object and select a filename.
2. Create some code and save the code to the filename.
3. Associate the Script with a Test (or possibly other MYNAH objects).

Section 2.2 describes in detail how to create script code.

2.1.2 Execution

You have several methods from which to choose for executing scripts. You can run scripts:

- Interactively from the UNIX command line (using **xmytclsh**)
- Interactively in the MYNAH GUI (using the Script Builder)
- Un-attended in the Background Execution Environment (using the GUI or CLUI).

Each execution of a script can produce a directory containing a record of the execution. In addition, if a script is described in a Script Object, then each execution may produce a database object called a Runtime Object and, for each Test Object associated with a Script Object, a Result Object.

Section 2.3 describes in detail how to execute script code.

2.2 Creating Scripts

The MYNAH System provides several means for creating scripts. You can:

- Use your favorite editor (e.g, **vi**)
- Use a MYNAH Script Object (code view)
- Use the interactive MYNAH GUI Script Builder.

> **NOTE —** Both the Script Object and the Script Builder methods let you enter code using an external editor. The MYNAH System will use whatever editor you specify in your preferences. If you do not specify an editor, the system will default to **vi**.

You may choose one method over the others or use a combination of methods.

Once you've created a script, you can execute it using any of the methods described in Section 2.3.

> **NOTE —** For hints about what to put in your scripts, see Section 3.

2.2.1 Using an Editor to Create Code

One way to create a script is to use your favorite editor. Simply type the Tcl commands and MYNAH extensions detailed in this guide and save them to a UNIX file. This method is recommended only for experienced users since it requires extensive knowledge of the system.

One of the primary reasons for using an editor is to edit an existing script. For example, if you have a script that automates an interaction with an application on one system and you want to use this script on another system, all you would have to do is copy the script and change a few lines to connect to the second system.

Once you've created the script, you can then execute the script using any of the execution methods.

2.2.2 Using the Script Object Code View to Create Code

> **NOTE —** See the *MYNAH System Users Guide* for detailed information on using Script Objects.

Another method for creating script code is using the MYNAH GUI **Script Object**. Using a Script Object, you can:

- Type code directly in the Script Object Code View window
- Insert any of the templates that are delivered with the MYNAH System or that you create
- Load any script you previously created
- Have the GUI invoke your favorite editor and type in the script
- Use any combination of the above.

However you decide to create script code, script code will appear in the Script Object Code View as it does in Figure 2-1.

Figure 2-1. MYNAH GUI Script Object Code View

When you create a the Script Object, you specify the path and name of a script file in the UNIX file system. If the UNIX file exists, the MYNAH System loads the file into the Script objects code view. If the UNIX file doesn't exist, the code view will be empty.

When you save a script using the Script Object, an element, called an object, is created in the MYNAH database that points to the script file. In addition, a file is created in the UNIX file system, or, if one already exists, will be overwritten if you choose to do so.

You can execute the script directly from the Script Object (if you save the script) or by using one of the other execution methods.

2.2.3 Using the Script Builder to Create Code

> **NOTE —** See the *MYNAH System Users Guide* for detailed information on using the Script Builder.

The GUI's **Script Builder** is useful when you need to record keystrokes from a 3270 or asynchronous session and/or add Tcl statements for other types of SUT connections, procedures, etc. In fact, this is the recommended method of creating and debugging scripts since it can minimize the guesswork you have over what Tcl commands or extensions to use or the correct syntax when you use it to emulate and record a connection to a SUT.

The Script Builder can be accessed in one of two ways:

- If you want to be connected to the MYNAH database, use the **Tools** menu on any MYNAH window
- If you do not want to be connected to the MYNAH database, start a standalone Script Builder.

When creating script code using the Script Builder, you can have the Script Builder emulate a 3270 or asynchronous connection to access a SUT. You can then perform all of the actions you want to use to automate or test the SUT. The Script Builder can be set to capture all of your keystrokes. When you do this, the Script Builder automatically enters the MYNAH Tcl extensions with the correct syntax. If the statements completely represent what you want, then you do not need to modify the script. However, the Script Builder generates default statements. Therefore, you might want to modify the default statements, such as if you need to specify a different time-out for a specific connection.

- Insert any of the templates that are delivered with the MYNAH System or that you have created
- Insert existing script code
- Insert script code from a MYNAH Script object. This loads the file to which the database object points

> **NOTE —** Note this method is available only if you are not using a standalone script builder.

- Have the GUI invoke your favorite editor and type in the script
- Enter or edit the code directly in the Code view window
- Any combination of the above.

However you decide to create script code, script code appears in the Script Builder Code View as it does in Figure 2-2.

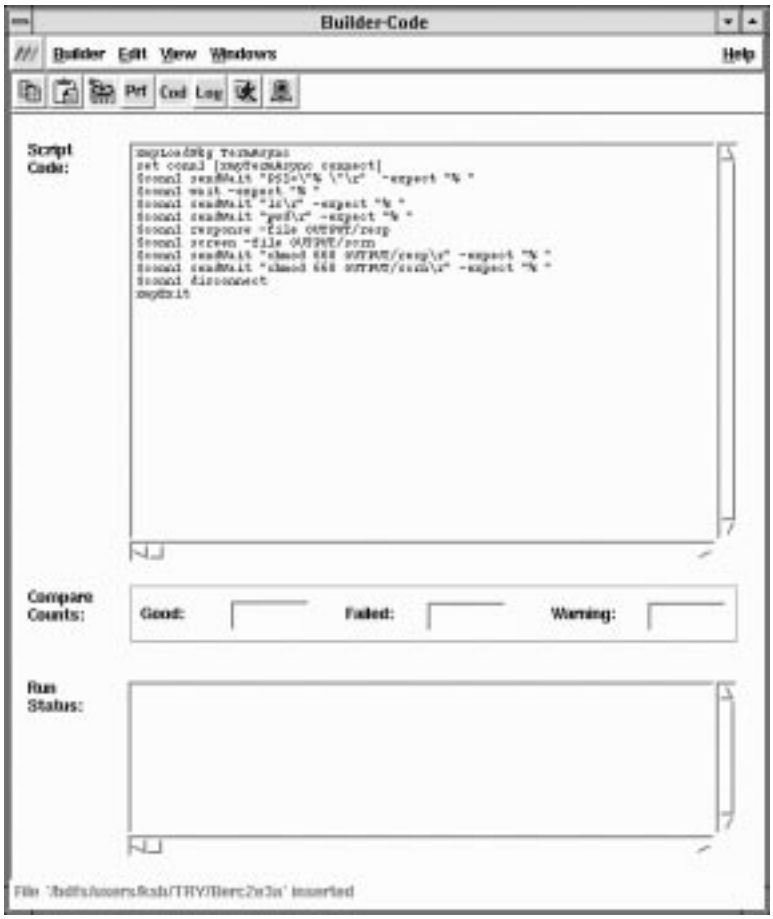

Figure 2-2. MYNAH Script Builder Code View

Until you save the code in a script, you can execute the code from the Script Builder's **Run** dialog only. If you save the code to a script, you can use any of the other execution methods.

2.3 Executing Scripts

There are several methods for executing scripts. Depending on how you use the MYNAH System, some methods may be more advantageous than others. The following sections describe these methods, detailing the steps for each method, and why you should or shouldn't use that method.

> **WARNING —** How you execute a script impacts various factors, such as UNIX environment variables and ownership of any output files. Section 3.3 discusses these issues and details a few methods of dealing with them.

When a script terminates, several things happen:

- Information is written to the appropriate output files. For more information, see Section 2.5.
- Appropriate database objects are updated (if the script was executed in database mode). For more information, see Section 2.6.

2.3.1 Using the Script Builder to Execute Code

Once you've used the Script Builder to load or create code, all you have to do to execute the code is select the **Run** option on the **Script Builder** menu or click on the **Run** button on the Script Builder Tool Bar.

> **NOTE —** For more information about the script builder see the *MYNAH System Users Guide*.

The Script Builder provides one method for executing script code. However, since the script builder does not create the MYNAH System's Test Management output objects (runtime and result objects), you should choose this methods when you do not need the Test Management capability. (for more information, see Section 2.6.)

You might choose to use the Script builder, for example, to re-run your newly created code (e.g, to verify that the code runs from beginning to end). Or, you might want to run existing code to investigate a "System Under Test" SUT problem.

When you use the Script Builder to execute code, there is an option called **Show Execution Progress**. If you select this option, as each line of code is executed, the code (and its Tcl return value) will appear in the dialog window shown in Figure 2-3.

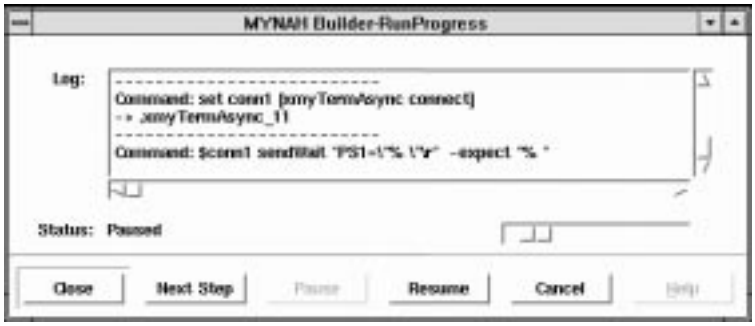

Figure 2-3. Script Builder Execution Progress Dialog

The Script Builder also has an option called **Display Remote Session Connections**. If you select this option and the script code requires a connection, the Script Builder will start a window emulating an asynchronous or 3270 synchronous window, such as the one in Figure 2-4. When a line of code is executed that sends a string to the system you are testing or automating, the string (and its response) will appear in the emulation window.

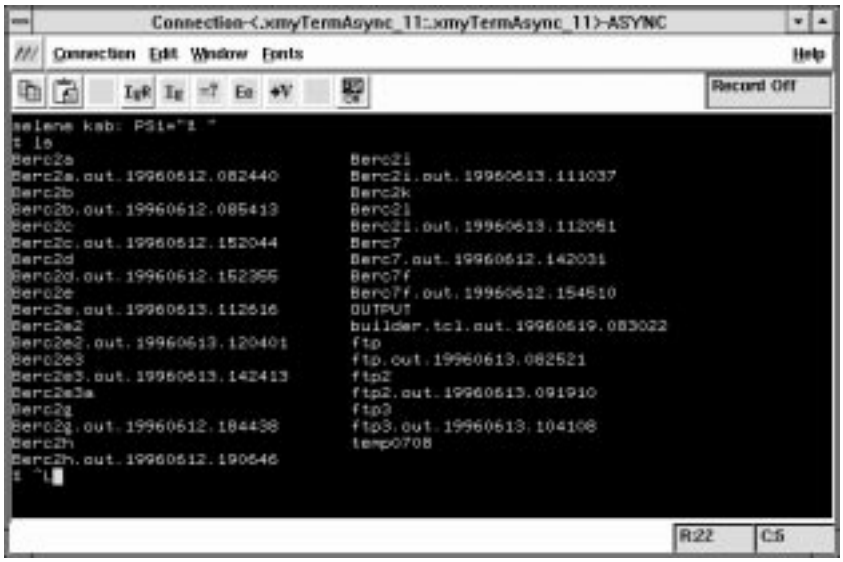

Figure 2-4. Script Builder Remote Connection Execution Window

You can have the system run the script automatically or you can have the system run the script step-by-step, pausing after each line of the script has been sent. This is very useful when debugging a script since you will be able to tell exactly where a problem occurred.

NOTE — The Script Builder uses a type of **Script Engine** (**SE**) called an **embedded SE**. An SE is the MYNAH process that runs code, and an embedded SE is an SE that runs code by graphically displaying the screens associated with Term3270 and TermAsync Packages.

2.3.2 Using Background Execution

The MYNAH System lets you execute scripts as background processes. That is, you can submit the script to the MYNAH System, which will schedule the script for execution. You can continue with other work while the script is running. Background execution also allows you to use the MYNAH System's Test Management capability.

We will first explain the concepts of background execution and then describe the steps used to execute scripts as background processes.

NOTE — Background execution uses a type of SE called a **background SE**.

2.3.2.1 Background Execution Overview

There are two primary background processes in the MYNAH System, the Script Dispatcher (SD) and the Script Engine (SE).

A **Script Dispatcher** manages Script Engines. An SD's primary function is to "dispatch" script execution requests to an SE. Part of the dispatching job is to manage **concurrency** of script execution, which is the ability to synchronize between concurrently executing scripts. All script requests that come to a SD process are managed as part of one concurrency group. There can be more than one SD in a MYNAH configuration.

A **Script Engine** process runs scripts. An SE must be running as part of an **SE Group** under the control of an SD. The managed SEs are organized into SE Groups.

The combination of SDs and SE Groups constitutes the MYNAH **Background Execution Environment** (BEE). The relationship between SDs and SE Groups is

depicted in Figure 2-5, which shows two SDs. Each SD dispatches scripts to multiple SE Groups. Each SE Group in turn consists of some number of SEs.

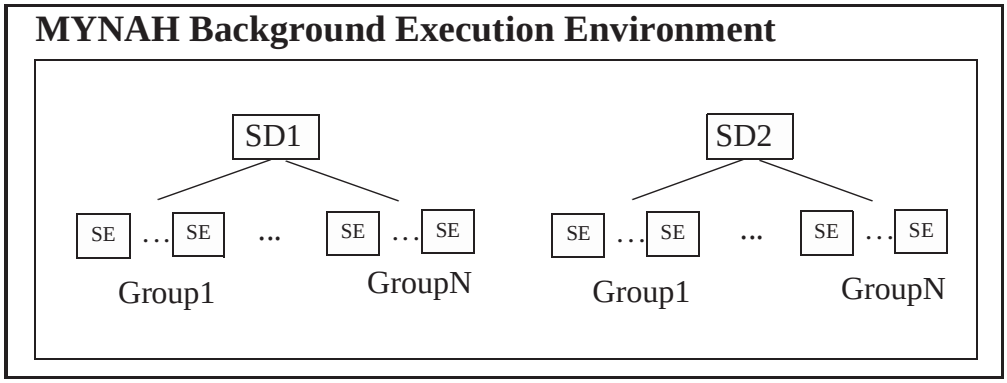

Figure 2-5. Background Script Execution Environment

A simple BEE is composed of one SD and a single SE Group.

You can use even this simple BEE to execute scripts in the background. This will provide you unattended execution and concurrency control.

The MYNAH System is delivered with a simple BEE already configured and ready to use. For the remainder of this section, we will assume you're using a BEE that includes one SD with one SE group and three SEs..

For a complete discussion on the power, extensibility, and flexibility of the BEE, see the *MYNAH System Administration Guide* and the *MYNAH System Users Guide*.

2.3.2.2 How to Submit Scripts to the Background

You can submit scripts to the background from either the command line or the GUI. For more information, refer to the *MYNAH System Users Guide.*

2.3.2.2.1 From the CLUI

You submit scripts to the BEE from the command-line by using the Command Line User Interface (CLUI) command **xmyCmd** and its sub-command **submit**. For example, if you have a script and a Script Object called *auto_ftp*, which you use to automate the retrieval of files from an anonymous **ftp** archive, you can execute it by typing

```
xmyCmd submit auto_ftp
```

The CLUI looks in the database to find the Script Object that is associated with your file (*auto_ftp*) and then sends a request to the BEE to execute the script associated with the

Script Object. If there is no Script Object associated with your file, the CLUI returns the message

```
cmyCmd(submit): Error cannot retrieve script
“<path>/auto_ftp” from database
```

If your script is not associated with a Script Object, you can execute the script using **-c** option, as in

```
xmyCmd submit -c auto_ftp
```

The **-c** option runs the script in the “cloaked” mode, meaning that no MYNAH database transactions are performed. The CLUI sends the name of your file (*auto_ftp*) to the BEE for execution but does not interact with the MYNAH database, for more information see Section 2.6.

2.3.2.2.2 From the GUI

To submit a script to the background from the GUI, you can use one of the database objects (e.g., script, test, sut). Simply choose the **Run** option and the selected Script(s) associated with the test will be sent to the BEE.

Database updates are always performed if a script is submitted from the GUI. The **Job Status** window is your tool to monitor the scripts you have submitted to the BEE.

2.3.3 Using xmytclsh

The MYNAH System provides a utility program, **xmytclsh** (Section 4.10), that lets you interactively run Tcl commands, including all MYNAH extensions.

The main reason to use **xmytclsh** is as a tutorial tool. While you’re learning the MYNAH scripting language, you can use **xmytclsh** to test the MYNAH commands and extensions. However, you can also use **xmytclsh** to execute existing script code.

This method, however, is not a very useful way of executing a script since it does not generate any of the MYNAH output files nor does it take advantage of MYNAH Test Management capabilities.

You could also type each line of the script directly into **xmytclsh**, but this is very tedious, and once you’ve finished you don’t have a reusable script.

To use **xmytclsh** to execute script code

1. Create the script, and save it to a file.
2. Start **xmytclsh**.
3. Use the **source** command (Section 4.11) to load the script.

NOTE — xmytclsh uses a type of SE called a **command-line SE**.

For example, if you have a script in a file called *logon*, you could execute it by typing the following:

```
xmytclsh
>source logon
```

assuming you are in the current directory containing the file *logon*.

2.4 SE States at Start Time

When an SE starts up, it runs in one of three modes: ***Stateless***, ***ConnOnly***, and ***FullState***. The mode cannot be changed after start-up. The following sections describe SE execution modes.

2.4.1 Stateless Mode

In the ***Stateless*** mode, when a script completes:

- All open connections are closed
- All Tcl file descriptors are closed
- The Tcl interpreter is deleted.

The Tcl start-up script is rerun each time a new interpreter is created. This means that any packages loaded with **xmyLoadPkg, (**for more information See Section 6.2.9**)** are reloaded.

You would use the ***Stateless*** mode when you do not want to maintain connections between scripts. For example, if you are submitting a series of scripts to different SUTs, you would include statements to log on and log off from the appropriate SUT in each script.

2.4.2 ConnOnly Mode

In the ***ConnOnly*** mode, when a script completes:

- Connections to the SUT are maintained
- All Tcl file descriptors are closed
- The Tcl interpreter is deleted.

You would use the ***ConnOnly*** mode when you want to maintain connections between scripts. For example, if you are submitting a series of scripts to the same SUT, you would include a statement to log on to the SUT in the first script only, include a statement to log off from the SUT in the last script only, and all scripts in between would assume that you are still logged on to the SUT.

2.4.3 FullState Mode

In the FullState Mode, when a script completes;

- Connections to the SUT are not maintained
- All Tcl file descriptors are not closed
- The Tcl interpreter is not deleted.

In the ***FullState*** mode, scripts submitted to an SE Group go to the first available engine in that group. No re-initialization is performed between each script execution.

When users submit scripts to an SE group running, they should ensure that the Tcl interpreter in each SE is in the same state (e.g., same variables defined, etc.,) otherwise scripts submitted to such a group could behave differently depending on which SE they run in.

> **NOTE —** Command-line SEs (i.e., those submitted using **xmytclsh**) are always in ***FullState*** mode, regardless of the *xmyConfig* file setting, since they run individual script lines, not entire scripts.

You would use the ***FullState*** mode when you are submitting individual scripts. Depending on your implementation of the MYNAH System and needs, each script would include statements to log on and log off from the appropriate SUT.

> **NOTE —** Symbol tables are ***not*** retained by SEs, even those running in ***FullState*** mode. Each time a script is executed, a new (possibly empty) symbol table is created containing the symbols passed to the engine in that execution request.

2.5 File Output

Whenever a script is executed, the SE may produce files in an output directory. Whether or not files are produced and how much output is produced are controlled by the **OutputLevel** entry. Users can control the amount of output that is produced from within a script using the **xmyVar(OutputLevel)** array variable Section 5.12. This variable is initially set by the **OutputLevel** parameter of the **Engine** entry in the *xmyConfig* file.

There are several types of output files: *output*, *SUTimage*, *compares*, *stdout*, and *stderr*. These types of output files are described below.

2.5.1 Determining How Many Output Directories to Retain

The number of output directories to retain is configured by the MYNAH Administrator in the *xmyConfig* file.

- If the MYNAH System has been configured so that zero backups be kept, any existing directories for the current script are removed before a new one is created.
- If a finite number of backups are being kept, then when the finite number of backups is reached the oldest output directory is removed after the current run completes.
- If unlimited backups has been requested, no removal takes place.

If an SE finds a directory with the same name and timestamp as the one it is trying to create, the SE sleeps a second, regenerates the directory name, and tries again, so as not to overwrite another SE's active output directory.

Subsequent runs increment the number of output directories unless the SE is configured to keep only one copy of each script's output.

2.5.2 Location of the Output Files

By default, the MYNAH System will generate output files in the directory containing the script. The output files are contained in a subdirectory with the same name as the script plus the suffix "out" and a timestamp.

For example, if the current date and time is September 1, 1996, 4:28:24 PM, the output directory for the current run of the script */home/scripts/script1.tcl* is */home/scripts/**script1.tcl.out.19960901.162824***.

> **NOTE —** You must open the UNIX permissions of the directory containing the script or the system will not be able to generate the output subdirectory. For example, set the permissions to ***775***.

2.5.2.1 Other Possible Locations

The default location of the script output directory can be overridden by the MYNAH Administrator setting either the **OutputRoot** or **OutputPath** entry in the *xmyConfig* file in the **Engine** section.

OutputPath specifies the script output directories. The hierarchy of the original scripts is not maintained; this is a flat directory. For example if OutputPath is set to

/MYNAH/data,
and an execution request for the script.

```
/HOME/scripts/script1.tcl,
is received, then the script output directory will be

/MYNAH/data/script1.tcl.out. <timestamp>
```

OutputRoot specifies the path for the root file system under which script output directories should be created. Script output directories are created using the same directory hierarchy as the original scripts. For example, if **OutputRoot** is set to

```
/MYNAH/data/outputs
```

/output, and an execution request for the script */home/scripts/script1.tcl* is received, the script output directory will be

/mynah/data/outputs/home/scripts/script1.tcl.out.<timestamp>.

2.5.3 Content of the Output Directory

A script output directory for a single script execution may consist of several files.

> **NOTE —** These files are created during execution, but if a file is empty after the execution has completed, the file is deleted. Thus, you may temporary see these empty files in your directory.

The *output* file	This file contains events as defined below Section 2.5.4. It is the highest level view of a script execution.
SUTimage.<*conn*> files	These files contain SUT input and output, where <*conn*> is the connection handle, e.g., *SUTimage .xmyTermAsync_1*. This output can be in the form of screens or messages. There may be one or more files, and these files are created only if a SUT is accessed. If sutimage events are not written to the *output* file, then output to the *SUTimage* files is also not logged.

The *stdout* and *stderr* files	These files contain anything that is coded to write to the standard output or error. These files are created during execution and are typically empty. If so, these files are removed upon completion of script execution.
The *compares* file	This file contains the differenced output for any compare commands, that is, if your script compares one event against one or more events, the expected and received output are written to this file. If compare events are not being written to the *output* file, then output to the *compares* file is also not logged.

For example, after executing a script called *abc.tcl*, you may see the following files and directory (assuming your output level included *sutimage*, *compare*, *stderr*, and *stdout* and your script contained compares, access to a SUT, and commands to write to *stderr* and *stdout*):

- In the script directory:

```
abc.tcl
abc.tcl.out.19960612.082440/
```

- Then in the output directory ***"abc.tcl.out.19960612.082440"***:

  SUTimage..xmy3270_1
  compare
  output
  stderr
  stdout

2.5.4 The Output File

The *output* file contains script output information formatted using well-defined, colon separated fields. The categories of script output are:

- child script (childscr) events
- compare events
- error events
- language trace events
- script events
- summary events
- sutimage events
- suttiming events
- test object (testobj) events
- user events.

The first five fields of every event line contain the following information:

```
<date>:<time>:<catagory>:<pkg>:<type>
```

where

- <***date***> and <***time***> are date and timestamp with format ***YYYYMMDD:HHMMSS***
- <***category***> is one of the listed categories (e.g., ***sutimage***)
- <***pkg***> is the MYNAH package producing the message (e.g., ***TermAsync***)
- <***type***> is the type of message, which differs for each category of output.

The following subsections describe the format of each category of output.

> **NOTE —** See Section 3.4 for information on setting output levels.

2.5.4.1 Child Script Events

Child script (childscr) events are logged when connecting to a remote SE and when any requests or replies related to child script execution occur.

Each childscr event line takes the form

```
<date>:<time>:childscr:<pkg>:<type>:<conn>:<msgId>:<script name>:<status>:<exit string>
```

where:

- <***pkg***> is ***general***
- <***type***> is one of ***send***, ***receive***, ***pause***, ***resume***, or ***cancel***
- <***conn***> is the name of the SE Group to receive the request
- <***msgId***> (***pause***, ***resume***, ***cancel***, ***destroy***, ***wait*** only) is the message object id
- <***script name***> is the name of the script to be executed remotely
- <***status***> is the result of the operation
- <***exit string***> (***receive*** only) is the Tcl result of script execution. (This string will be truncated if it exceeds the allowable length for output file lines)

The **sendWait** command (Section 7.1.6) creates two log messages:

- ***send***, which lists the child script that was sent and whether the **send** succeeded
- ***receive***, which lists the actual response

Examples

```
19960529:190912:childscr:general:send:SeGp2::/u/kjd/child01.tcl:channel send succeeded
19960529:190914:childscr:general:receive:SeGp2::/u/kjd/child01.tcl:TCL_OK:
```

2.5.4.2 Compare Events

Compare events detail the differenced output for any MYNAH language **compare** commands that automatically update the **xmyVar (GoodCompares), xmyVar(FailedCompares)** and **xmyVar(WarningCompares)** counts. These are:

- General package
  - **— xmyCompare**
  - **— xmyDiff**
  - **— xmyRegex**
- TermAsync package
  - **— $connection compare**
- Term3270 Package
  - **— $connection compare**
- App-to-App Package (xmyFCIF)
  - **— $handle compare**
  - **— $handle compareTags**
  - **— $handle extraTags**

Each compare event line takes the form

```
<date>:<time>:compare:<pkg>:<type>:<label>:<result>:<index>
```

where:

- **<*pkg*>** is one of ***General***, ***TermAsync***, ***Term3270***, or ***FCIF***
- **<*type*>** is one of ***data***, ***fcif***, ***diff***, ***screenRegion***, ***regex***, or ***string***
- **<*label*>** is non-null if the compare specified a label
- **<*result*>** indicates whether the test passed or failed by specifying one of ***good***, ***failed***, or ***warning***.
- **<*index*>** indicates the number of bytes into the *compares* file to look for the record of that **compare**. The numbers are indexes into the *compares* file in the output directory

Example

```
19951201:090000:compare:3270:screenRegion::good:226
```

The MYNAH comparison commands log compare events to the *output* file and also write expected and actual data, if possible, to the *compares* file. (If an expression is being evaluated instead of data being compared, the expression is written to the *compares* file.)

2.5.4.3 Exception (error) Events

The script engine logs an exception event when an exception occurs in the Tcl interpreter. An exception event contain the value of ***errorCode*** and ***errorInfo*** as set by the interpreter and take the form

```
<date>:<time>:error:<pkg>::<message>
```

where:

- <***pkg***> is ***errorCode*** or ***errorInfo***
- <***message***> is the error message itself

Example

```
19960613:112054:error::errorInfo:Connection went down
19960613:112054:error::errorInfo: while executing
19960613:112054:error::errorInfo:"$conn1 wait -expect "$ ""
19960613:112054:error::errorInfo: (file "/u/kjd/ex.tcl" line 4)script: /u/kjd/ex.tcl line: 4
19960613:112054:error::errorCode:NONE
```

2.5.4.4 Language Events

Language events record the lines of your scripts, with the format

```
<date>:<time>:lang:<pkg>:command:<command>
```

where:

- <***pkg***> is the language package that is used
- <***command***> is the Tcl command itself. Variables with unprintable characters or lines longer than 512 characters are truncated.

Examples

```
19960613:142414:lang:tcl:command:xmyLoadPkg TermAsync
19960613:142414:lang:tcl:command:set xmyVar(OutputLevel) {sutimage, lang, script}
19960613:142414:lang:tcl:command:xmyTermAsync connect
19960613:142414:lang:tcl:command:set conn1 [xmyTermAsync connect]
19960613:142414:lang:tcl:command:$conn1 sendWait "PS1=\"% \"\r"  -expect "% "
19960613:142415:lang:tcl:command:$conn1 wait -expect "% "
19960613:142415:lang:tcl:command:$conn1 sendWait "ls\r" -expect "% "
19960613:142415:lang:tcl:command:$conn1 sendWait "pwd\r" -expect "% "
```

2.5.4.5 Script Events

Script events record high level script activity, e.g., when the script began execution. Script events have the format

```
<date>:<time>:script:<pkg>:<type>:<group>:<exit status>:<exit string>
```

where:

- <***pkg***> is null
- <***type***> is one of ***start, stop, cancel, pause, resume,*** or ***abort***
- <***group***> is the SE Group to which this SE (the one running the script) belongs
- <***exit status***> is the return code from the Tcl interpreter for types ***stop***, ***cancel***, and ***abort*** only
- <***exit string***> is the possibly truncated Tcl result of script execution for types ***stop***, ***cancel***, and ***abort*** only

Examples

```
19960613:161728:script::stop:SeGp1:TCL_OK:
19960809:171420:script::stop:SeGp1:TCL_OK:Compares OK
```

2.5.4.6 Summary Events

Summary events are written to the output file at script completion time. They include the system variables, the results that are sent to the database, and the symbol table. They have the format

```
<date>:<time>:summary:general:var:xmyVar(<name>) <value>
<date>:<time>:summary:general:results:<results list split into 80 char lines>
<date>:<time>:summary:general:symtbl:<symbol table split into 80 char lines>
```

Examples

```
19960613:161728:summary:general:var:xmyVar(Channel)     = xmySE0007SD1
19960613:161728:summary:general:var:xmyVar(DatabaseMode)        = 1
19960613:161728:summary:general:var:xmyVar(EngineMode)  = stateLess
19960613:161728:summary:general:var:xmyVar(EngineType)  = background
19960613:161728:summary:general:var:xmyVar(ExitHandler) =
19960613:161728:summary:general:var:xmyVar(FailedCompares)      = 0
19960613:161728:summary:general:var:xmyVar(GoodCompares)        = 1
19960613:161728:summary:general:var:xmyVar(LibraryPath) = /home/mynah/lib
19960613:161728:summary:general:var:xmyVar(OutputLevel) = error
19960613:161728:summary:general:var:xmyVar(RuntimeId)   =
19960613:161728:summary:general:var:xmyVar(SEGroup)     = SeGp1
19960613:161728:summary:general:var:xmyVar(SubmittedBy) = ksb
19960613:161728:summary:general:results: {result {}}
19960613:161728:summary:general:results: {goodCompares 1}
19960613:161728:summary:general:results: {failedCompares 0}
19960613:161728:summary:general:results: {warningCompares 0}
19960613:161728:summary:general:symtbl:<symbol table empty>
```

2.5.4.7 Sutimage Events

Sutimage events are produced for only the Term3270, TermAsync, and App-to-App Packages. If sutimage events are being produced, then each time a screen or message is sent or received a sutimage event is produced having the format

```
<date>:<time>:sutimage:<pkg>:<type>:<conn>:<opt>:<index>:<length>
```

where:

- <***pkg***> is the name of the MYNAH package producing the sutimage
- <***type***> is one of ***snd***, ***rcv***, or ***state***. (***state*** is used to produce a record of the "current" screen. It is used by the TermAsync package only.)
- <***conn***> is the handle of the connection
- <***opt***> is an optional field used by the domain to indicate the cause of the sutimage
- <***index***> is a pointer to the start location of the sutimage in the *SUTimage* file
- <***length***> is the length (in domain-specific units) of the image

Examples

```
19960613:161727:sutimage:TermAsync:snd:.xmyTermAsync_37::2207:3
```

Each package may add additional fields as needed.

If you run code in background (i.e., from the Script Builder) and you do not select the **Generate Script Output** option on the Script Builder, then, in the Script Builder's **Log** view, the sutimage event will have an ***index*** of -1 and a ***length*** of 0, for example

```
19960914:115437:sutimage:TermAsync:snd:.xmyTermAsync_4::-1:0
19960914:115437:sutimage:TermAsync:rcv:.xmyTermAsync_4::-1:0
```

2.5.4.8 SUT Timing (suttiming) Events

SUT timing events are produced for only the Term3270, TermAsync, and App-to-App Packages. Timing events are logged each time a message is received from the SUT. They have the format

```
<date>:<time>:suttiming:<pkg>::<conn>:<command>:<send time>:<seconds>
```

where:

- <***conn***> is the connection handle
- <***command***> is the Tcl command that sent data to the SUT
- <***send time***> is the time of the last send in readable format
- <***seconds***> contains the number of seconds elapsed since the last send

Examples

```
19960613:161726:suttiming:TermAsync::.xmyTermAsync_37::06/13/96 16:17:26:0:10
```

2.5.4.9 Test Object Events

Test object (testobj) event lines are produced by the **xmyBegin** (Section 6.2.2) and **xmyEnd** (Section 6.2.6) commands. Test object event lines take one of these two forms

```
<date>:<time>:testobj:<pkg>:begin:<id>
<date>:<time>:testobj:<pkg>:end:<id>
```

where:

- <***pkg***> is null
- <***id***> is the test object id or test block label

Examples

```
19960809:171419:testobj::begin:2307
19960809:171421:testobj::end:2334
```

2.5.4.10 User Events

User events are produced on a per user basis. The MYNAH System provides a method for script writers to generate their own events to the output file using the **xmyPrint** command (Section 6.2.11). They have the format

```
<date>:<time>:user:<pkg>:<type>:<text>
```

where:

- <***pkg***>, <***type***> and <***text***> are provided by you through the MYNAH **xmyPrint** command.

User events can be used to surround processing statements with comments that help you in analyzing execution results.

Examples

```
19960611:145636:user:::MAIN
19960611:145636:user:::LEAVING PROC inits (/users/kjb/scripts/parent/par002.tcl)
```

2.5.5 SUTimage files

The Term3270, TermAsync, and App-to-App Packages produce one or more *SUTimage* files, which contain images of all strings, commands, etc., your script sends to a SUT and the SUT's responses. This output can be in the form of screens or messages. For example, the *SUTimage* file will contain an image of the asynchronous or synchronous (3270) screen as it appears after each send and receive.

A *SUTimage* filename takes the form *SUTimage*<*conn*>, where <*conn*> is the connection handle, e.g., *SUTimage..xmyTermAsync_1*.

One *SUTimage* file is generated for each connection.

Each *SUTimage* entry takes the form

```
IMAGE HEADER - <type> (index:<index_number>)
Body
IMAGE FOOTER -
```

where :

- **<*type*>** is one of ***String Sent***, ***Response***, or ***Screen***. These are equivalent to the output file sutimage events **<*type*>** entries ***snd***, ***rcv***, and ***state***, respectively.
- **<*index*>** is a pointer to the start location of the sutimage. This is the same index number as the output file sutimage events **<*index*>** entry.
- ***Body*** is the string sent to or received from the SUT or the contents of the current screen.

The following are examples of the types of *SUTImage* file entries that the TermAsync Package produces:

```
IMAGE HEADER - String Sent (index:0)
PS1=”% “^M
IMAGE FOOTER -

IMAGE HEADER - Response (index:123)
PS1=”% “^M
$ %
IMAGE FOOTER -

IMAGE HEADER - Screen (index:2636)
PS1=”% “
$ % ls
Debug_ksb                              mytrace2
debug_s10                              mytrace2b
ioctelnet.log                          temp1
mytrace
xmyDiff.tcl.out.19960506.143429
%









IMAGE FOOTER -
```

The following is an example of the types of *SUTImage* file entries that the Term3270 Package produces:

```
IMAGE HEADER - Screen (index:4003)
   AUTHORIZED USE ONLY   -  IFS STU08

 DATE:  05/31/96     TIME: 13:21:53

 NODE NAME: TCP10081

 USERID:

 PASSWORD:

 USER DESCRIPTOR:
 GROUP NAME:
 NEW PASSWORD:

    OUTPUT SECURITY AVAILABLE




IMAGE FOOTER - Screen
```

2.5.6 compares File

The *compares* file contains the differenced output for any of the **compare** commands listed in Section 2.5.4.2.

Each of these commands will also produce **compares** blocks in the *compares* file.

Each *compares* entry takes the form

```
COMPARE HEADER - <conn> compare (index:<index_number>)
Body
result: <result>
COMPARE FOOTER - <conn> compare
```

where:

- **<*conn*>** is the connection handle, e.g., ***.xmyTermAsync_11***, if any.
- **<*index_number*>** is the number of characters into the *compare* file where the compare begins. This is the same index for the *output* file Compare events.
- ***Body***
  - — Contains any expression you entered as an argument to **xmyCompare** (Section 6.2.3).
  - — Lists, in alternating rows, the expect and receive responses from domain instances. This only occurs if the compare failed.
- **<*result*>** specifies if the **compare** failed, succeeded, or warned of a potential failure. This can be one of the following:
  - — 1 (good)
  - — 0 (failed)
  - — 0 (warning)

The following are examples of each type of *compares* file entries:

This comparison was successful, so the system did not generate expected or received statments.

```
COMPARE HEADER - .xmyTermAsync_11 compare (index:0)
result: 1 (good)
COMPARE FOOTER - .xmyTermAsync_11 compare
```

This comparison failed, so the system generated expected or received statements.

```
COMPARE HEADER - .xmyTermAsync_11 compare (index:112)
1:expected:<1999>
1:received:<1996>
result: 0 (failed)
COMPARE FOOTER - .xmyTermAsync_11 compare
```

This comparison, the result of an unevaluated expression, generated a warning.

> **NOTE —** When a comparision is generated as the result of an evaluated or unevaluated expression, the same information, with the exception of the result of the comparision, is generated whether the comparision was good, failed, or warning.

```
COMPARE HEADER - xmyCompare (index:224)
expr: $childvar3 == "333"
result: 0 (warning)
COMPARE FOOTER - xmyCompare
```

2.6 Database Output

Database output is only produced when a script is associated with a Script Object. Database output consists of the Runtime Object and possibly Result Objects.

2.6.1 Runtime Objects

One Runtime Object is produced for each run of a script associated with a Script Object. The Runtime Object records information about the executed script such as

- The script name
- Who submitted the script
- When script execution started and ended
- The SD and SE Group that executed the script
- Where the output from the execution is located
- The execution status
- A summary of the execution, e.g. any errors that aborted the execution.

Runtime Objects can be accessed from the GUI's Database Browser, Job Status window, or a Script Object Run History View. For more information on Runtime Objects, see the *MYNAH System Users Guide*.

Several fields are updated in the Runtime Object:

> **NOTE —** These fields are visible in the Job Status Object.

- **Run Status** — This is set to either completed, aborted, or canceled
  - — completed - ran to some normal termination
  - — aborted - run aborted due to execution problem (for example, "cannot open outputDir")
  - — canceled - script execution was canceled before completion
- **Run Summary** — This is set to the result of the last Tcl statement that was executed. For example, if the last statement executed in a script was

  ```
  xmyExit "success"
  ```

  then the Run Summary field will be set to "success". See Section 5.5 on the **xmyVar(ExitHandler)** variable for further discussion on how script writers can control the value of Run Summary.

- **Tcl Code Status**— This is set to the exit code of the Tcl interpreter. The values can be:
  - — N/A- script never started executing
  - — TCL_OK - script exited without a Tcl error
  - — TCL_ERROR - a Tcl error was produced. The most common reason for this is a syntax error in the Tcl code.

2.6.2 Result Objects

Result Objects record actual results for particular Tests. One or more Result Objects may be produced each time a script is executed.

Result Objects are produced if:

- The script code contains **xmyBegin** (Section 6.2.2) and **xmyEnd** (Section 6.2.6) statements. In this case one Result Object will be created for each **xmyBegin**/**xmyEnd** pair.
- If there are no **xmyBegin**/**xmyEnd** statements but the script object is associated with one and only one Test Object.

  In this case one Result Object will be created for the associated Test.

Result Objects are accessed from the GUI's Database Browser or from a Test Object Results view. For more information on Result Objects, see the *MYNAH System Users Guide*.

If a Script is associated with only one Test or if a Script has **xmyBegin** (Section 6.2.2) and **xmyEnd** (Section 6.2.6) statements, then the SE will create one or more Result Objects. The SE fills in Total counts for Successful, Warning and Failed compares. These values come directly from **xmyVar** values.

The SE also provides values for Activity State, Test Status, Status Source and Reason Code. For a complete discussion about these fields, please see the *MYNAH System Users Guide*.

2.7 Execution Without Database Update

You may create and run script code even if you don't create a Script Object or if you have a Script Object but you don't want the database to be updated (such as when you are automating a task).

You can do this in one of three ways:

- Use the **xmyCmd submit** command with the **-c** option
- Run the code interactively in the Script Builder

- Run the code interactively using **xmytcsh**

> **NOTE —** For more information about these methods, refer to the MYNAH Users Guide.

2.8 Loading Procedures

A Tcl procedure is a script that is used to invoke a command, letting you reuse the script. See Section 4.7 for information on procedures.

A script can access (i.e., can call) procedures using any of the following methods:

1. Procedures can be defined locally within a script, for example, the following procedure will add two numbers together:

   ```
   > proc sum { a b } { return [expr $a+$b] }
   ```

   This procedure can later be called, as in

   ```
   > sum 2 3
   5
   ```

2. You can save the procedures to a file and then load them into your script using the **source** command (Section 4.11).

   > **NOTE —** You can save more than one procedure in each file.

   For example, you could save the addition procedure in Item 1 to a file, e.g., *myprocs.tcl*, and load it by typing

   ```
   source myprocs.tcl
   ```

   in your script. You can then use the addition procedure by typing

   ```
   > sum 3.45 8.53
   11.98
   ```

3. You can use any autoloadable procedures. These are procedures that are searched for by the Tcl interpreter the first time they are called in a script. The search path can be set for the **ProcRepository** parameter in the MYNAH configuration file, *xmyConfig*.

   The default value for the **ProcRepository** parameter is *$XMYDIR/lib/tcl:$XMYHOME/lib/tcl*. The setting of this field affects the setting of the standard Tcl variable *auto_path*. A script writer with procedures defined in other directories can add paths to the *auto_path* variable directly in the script.

> **NOTE —** See Appendix A.30 for more information on creating procedure libraries and global variables that are defined or used by the procedures in the Tcl library.

The **Insert Template** dialog on the MYNAH GUI's Script Builder and Script Object displays all procedures that are found in the procedure libraries. (See the *MYNAH System Users Guide*.)

Administrators can override what is displayed in the **Insert Template** dialog by manually creating template files. By default, however, bringing up **Insert Template** and changing **Type** to **Procedure** will display all procedures the Tcl interpreter can automatically load (i.e., those in *$XMYDIR/lib/tcl:$XMYHOME/lib/tcl*).

> **NOTE —** Procedures libraries in directories not specified in the ProcRepository parameter in the *xmyConfig* file ***do not*** show up in Insert Template.

3. Scripting Hints

This section contains hints that you may find useful as you create your scripts.

> **NOTE —** This section will often use MYNAH scripting terms that will be explained in later sections. You may wish to come back to this section after you have read these later sections.

3.1 Concealing Sensitive Data

For security reasons, sensitive data, such as passwords, is usually concealed. The MYNAH System provides several means of entering sensitive information.

3.1.1 Prompting for Sensitive Data using the Script Builder

You can use the **xmyPrompt** command (for more information on xmyPrompt see Section 6.2.12) to prompt you (or another user) to enter a password.

> **NOTE —** You can only use **xmyPrompt** when you are running code in the MYNAH GUI's interactive Script Builder since this command requires the use of an embedded SE, which is only available when using the Script Builder.

For example, if you enter the following code in the Builder

```
keylset psswd -prompt "Enter Password" -echo false
set result [xmyPrompt [list $psswd]]
```

The GUI will display the window in Figure 3-1 when you run the script from the Script Builder and the script reaches the **xmyPrompt** command.

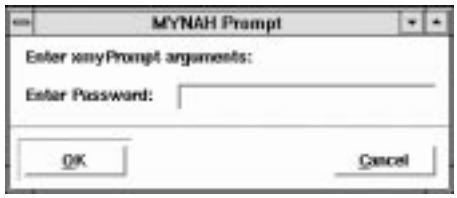

Figure 3-1. xmyPrompt Window

xmyPrompt takes as its input the keyed list created by the **keylset** command. The data that you enter is saved to the variable *$result*. The variable *$result* can be used in a subsequent command, e.g.,

```
$conn1 type -position “9 12” -text $result
```

to enter the password in the password field.

> **NOTE —** See Appendix B.11.4 for information on the **keylset** command.

3.1.2 Obtaining Sensitive Data for Scripts That Run in the Background

To allow you to pass sensitive data to the background SE, the MYNAH System provides the means for encrypting and loading data stored in files.

The **xmyUdb** command lets you load and save sensitive data into your scripts. These files can be used for databases or to hold such sensitive material as passwords. For security reasons, the files can be encrypted using the **des** system software, which is part of the Encryption Kit supplied along with the operating system, see Administration Guide for details. For added security, the key used to decrypt the encyrpted file can be scrambled using the CLUI command **xmyCmd scramble**.

> **NOTE — des** is only available in the United States. Since the MYNAH utilities use **des** for encryption and decryption, the methods described in this section will only work in the U.S.

A **clear tag-value database** is an ASCII file containing two columns separated by spaces or tabs. The first column contains the tag, and the second column contains the values. For example, if you wanted a file containing a list of login ids, you could enter them as in the following:

```
#Tags   Ids
1stId   kjd
2ndId   ksb
3rdId   lsmyth
```

Lines beginning with a pound sign (#) are treated as comments. This lets you label columns, as we did in the previous list of login ids.

An **encrypted** database is simply a clear database that has been encrypted using **des**.

The MYNAH System provides several utilities that let you process encrypted material via your scripts.

3.1.2.1 xmyUdb

The **xmyUdb** command (see Section 6.3.1) lets you load a tag-value file into a Tcl script as a keyed list that can be accessed and modified using the standard keyed list functions, e.g., **keylget** and **keylset**, etc. If you've encrypted the file, you can have **xmyUdb** decrypt it. Conversely, you can have **xmyUdb** write keyed lists to a tag-value file, and, if you desire, encrypt the file.

> **NOTE —** For more information on keyed lists, see Appendix B.11.

3.1.2.2 xmyCmd scramble

The **scramble** sub-command to the MYNAH CLUI command **xmyCmd** (see Section 6.3.2 of this guide and the *MYNAH System Users Guide*) scrambles the generated key that is used by **xmyUdb**. This lets you safely leave the keys in flat files (like Tcl scripts). Scrambled keys can be entered in the **Engine** entries in the MYNAH *xmyConfig* system configuration file or provided when loading or saving sensitive data. The SE internally unscrambles the key and uses it to decode encrypted tag-value files.

3.1.2.3 Encrypted Database Files using des

To create an encrypted database, follow these steps:

1. Create a clear database file.
2. Type the command:

   `des -e` *`input-file output-file`*

   where

   input-file is the name of the clear database file and ***output-file*** is the name to be given to the encrypted database file.
3. When you are prompted for a key that **des** should use to do the encryption, enter a key (eight characters or fewer).

   > **NOTE —** Be sure to make a note of the key that you use for the encryption. ***You won't be able to decode the encrypted file without the key***.

   For example, to encrypt a file called *pass*, which contains passwords, save it as a file called *pass.e*, and use a key ***enter***, you could type

   ```
   des -e pass pass.e
   ```

   The following will appear, prompting you to enter the key:

   ```
   Enter key:
   ```
4. Delete the clear database file.

You can enter the key directly as an argument to **des**, as in the following:

```
des -e -k enter pass pass.e
```

For a MYNAH script to decrypt the encrypted file that you just created, it must know the key that you used.

There are several ways to provide the key. The recommended order of these methods is as follows:

1. Have the MYNAH Administrator enter the key (scrambled or unscrambled) as an argument to the **Key** parameter for an **Engine** entry in the *xmyConfig* file. For example, if the key has been entered as an argument to the **Key** parameter, you would enter a line such as the following:

   ```
   set ids [xmyUdb read -file pass.e -decrypt]
   ```

   > **NOTE —** Anyone who knows the key, however, can decode the encrypted file and read its contents. For that reason, we recommend you use **xmyCmd scramble** to scramble the key.

2. Enter the key (scrambled or unscrambled) directly as an argument for **xmyUdb**, for example

   ```
   set ids [xmyUdb read -file pass.e -decrypt -key enter]
   ```

3. If you're running a script from theScript Builder, you can enter the **xmyPrompt** command in the script and have the script prompt you for the key, for example

   ```
   keylset key_prompt -prompt "Enter des Key" -echo false
   set des_key [xmyPrompt [list $key_prompt]]
   set ids [xmyUdb read -file pass.e -decrypt -key $des_key]
   ```

   The window in Figure 3-2 will appear.

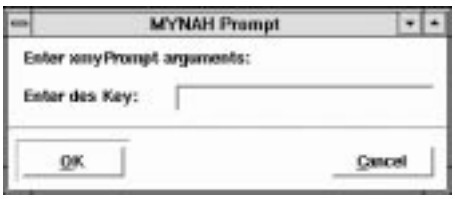

Figure 3-2. Example Window Prompting for des Key

3.1.2.4 Using an Encrypted File

One of the main uses of encrypting a file and loading it into a MYNAH script would be to load passwords. When using the GUI, you can use the **xmyPrompt** command to prompt you for the password, but you can't do this if you run a script from the command-line (using **xmytclsh**) or from the background (using **xmyCmd submit**).

What you can do is enter the password in a tag-value file and encrypt it. In fact, you can use the same file to load the login id, too.

3.1.2.4.1 Working With Keyed Lists

Before we begin, let's look briefly at how you work with a keyed list.

This is the list of ids we mentioned in Section 3.1.2.

```
#Tags   Ids
1stId   kjd
2ndId   ksb
3rdId   lsmyth
```

We saved this list to a file called *IDS*. Using **xmyUdb**, we can load this file as a keyed list.

```
set ids [xmyUdb read -file $env(HOME)/data/IDS]
{1stId kjd} {2ndId ksb} {3rdId lsmyth}
```

> **NOTE —** Notice that beginning the line
>
> `#Tags Ids`
>
> with a pound sign commented this line. Therefore it did not appear in the return from the **xmyUdb** command.
>
> This line was added only to illustrate the contents of each column, labeling each column.

Using the **keylget** command (Appendix B.11.2) we can extract values from this list by specifying the tag.

```
set id1 [keylget ids 1stId]
kjd
```

keylget searches the keyed list for the tag we specified, in this case ***1stId***, and extracts the value associated with it, ***kjd***.

Using the same techniques, you can extract values from an encrypted file simply by adding the **-decrypt** argument and the decryption key, as described in Section 3.1.2.4.2.

3.1.2.4.2 Example of Loading Data from Keyed Lists

For our example, we'll use the login id and password for an anonymous **ftp** session.

> **NOTE —** While the password for an anonymous **ftp** session (usually your e-mail address) is not sensitive data, the techniques we'll use will apply for cases where the password is sensitive.

1. Create a flat file — let's call it *login* — containing the id and password as tag-value pairs, one pair per line, such as:

   ```
   #Tag      Value
   1stId     anonymous
   1stPass   kjb@
   ```

2. Encrypt the file *login* (with the key ***open***) and save it as a file called *login.e*.

   ```
   des -e -k open login login.e
   ```

3. Delete the *login* file.

4. If you want, you can scramble the key ***open***. In our example we're saving it to a file called *key1*.

   ```
   xmyCmd scramble -k open key1
   ```

   The output file is optional, but this way you have a permanent record of the scrambled key. Even if someone else gains access to this scrambled key, they would not be able to use it to decrypt your file; only an SE can use the scrambled key to decrypt the file when it is running a script.

5. You can now load this file, as in the following.

   ```
   xmyLoadPkg TermAsync
   set conn2 [xmyTermAsync connect]
   set logon [xmyUdb read -file $env(HOME)/data/login.e \
              -decrypt -key [read file key1]]
   set logid [keylget logon 1stid]
   set password [keylget logon 1stpass]
   $conn2 sendWait “ftp fake_ftp.com\r” -expect “: “
   $conn2 sendWait “$logid\r” -expect “d:”
   $conn2 sendWait “$password\r” -expect “> “
   $conn2 sendWait “cd pub/new_this_week\r” -expect “> “
   $conn2 sendWait “prompt\r” -expect “> “
   $conn2 sendWait “lcd /u/kjb/NEW\r” -expect “> “
   $conn2 sendWait “mget *\r” -expect “> “
   $conn2 sendWait “bye\r” -expect “: “
   $conn2 disconnect
   xmyExit
   ```

3.1.3 Concealing Sensitive Data in Async SUTimages Files

You can conceal sensitive data in an Async *SUTimages* file by using the **-secret** option for a **send** (See Section 8.4.1.10) or **sendWait** (See Section 8.4.1.11) method. When you use this option, the string that is sent to the SUT will not appear in the *SUTimages* file associated with that connection. In its place, the string <*hidden data*> will appear.

> **NOTE —** See Section 2.5.5 for more information about the *SUTimages* file.

For example, you are logging onto a SUT and you must enter your password. If ***moses123*** is your password, you would code the following (with additional options that you would want and the appropriate string for the **-expect** option):

```
$conn5 sendWait “moses123\r” -expect “\] “ -secret
```

After you replay the script or code that contains this command, the *SUTimages* file associated with **$conn5** will contain the following information (note that index shown below is an example):

```
IMAGE HEADER - String Sent (index:17014)
<hidden data>
IMAGE FOOTER -
```

3.2 Debugging - Dealing with Errors and Exceptions

3.2.1 Overview

Tcl errors can occur for any number of reasons. You may have mis-typed and entered a nonexistent command. You may not have entered the correct number of arguments. There may be input/output problems. A command may have timed-out. Often, these errors are severe enough to abort processing.

Errors, though, are just part of the Tcl concept of **exceptions**. An exception is any event that can abort a script, be it an error or a command such as **break**, **continue**, and **return**. Tcl has the ability to "catch" exceptions, only part of the script is lost and the script can try to ignore or, if possible, recover from the exception. Using the return value from **catch**, you can code your script to manage the exception condition, e.g., decide whether to continue or gracefully stop a script at that point. The following subsections describe several methods for managing exception conditions.

3.2.2 Tcl Error/Exception Information Procedures

Tcl contains several commands related to exceptions, **catch** (Section 4.6.1), **error** (Appendix A.11), and **return** (Appendix A.48). The Tcl command that manages exceptions is **catch**, which can trap the error so that the script does not abort. **error** and **return** do not trap exceptions, but can be used when processing exceptions.

In addition, Tcl contains several global variables, **errorCode** (Section 4.6.2.1) and **errorInfo** (Section 4.6.2.2), that return information resulting from errors.

3.2.3 MYNAH Exception Handling

> **NOTE —** This section contains excerpts from a script using the Term3270 Package. The commands and methods for the Term3270 Package are used to illustrate using the exception handling processes. For a description of the Term3270 Package, see Section 9.

In general, MYNAH Tcl commands use the standard exception handling facility. If an exception is occurs (the command callback returns **TCL_ERROR**) and if not caught by the script, then the script aborts and the Tcl interpreter is deleted if the SE is running in **Stateless** or **ConnOnly** mode.

3.2.3.1 General Actions

Catching exceptions is done using the Tcl **catch** command.

If a MYNAH or Tcl command fails, the **catch** command

1. Creates a meaningful error message, which can be retrieved if you specify the ***varname*** argument for the **catch** command.
2. Populates the global Tcl variable **errorCode**. The variable consists of a list with one or more elements, the first of which is the class of error. The second argument is the symbolic name for the particular error. (e.g. **ARGMISSING.**)

   NOTE — errorCode is not implemented for all MYNAH extensions.

3. Returns **TCL_ERROR**.

3.2.3.2 Error Processing

When an error occurs, the current command is aborted. If this command is a child process of a script or procedure (i.e., is part of a larger script), the parent script or process is also aborted, and so on until all active Tcl commands are aborted. A message describing the error is generated and is available via the **errorInfo** and **errorCode** variables, but these may not tell all you need or want to know about where the problem occurred. You may want to generate a specific error message describing what happened and where. In addition, there may be situations where you want to continue script execution after an error has occurred.

In MYNAH 5.2, all error processing should be done via Tcl statements. Wait until a desired response occurs. If it doesn't occur, then you can use **catch** to evaluate the result and use the result of the **catch** command as the input for a way of gracefully exiting from the script.

Figure 3-3 shows an excerpt from a Term3270 package script where you are preparing to logon to a program.

```
set expscreen USERID
$conn type -text "stu08"
if { [catch {$conn sendWait -key enter -expect $expscreen}]} {
    xmyPrint "ERROR: did not get screen: $expscreen"
    syncImokLogoff
    $conn disconnect
    return 0
}
#if get here, have logon screen, enter logid

$conn type -text $logname
```

Figure 3-3. Error Processing Example

A facsimile of the logon screen appears in Figure 3-4.

```
AUTHORIZED USE ONLY   -  IMOK STU08
DATE:  05/31/96     TIME: 13:21:53
NODE NAME: TCP10081
USERID:
PASSWORD:
USER DESCRIPTOR:
GROUP NAME:
NEW PASSWORD:
   OUTPUT SECURITY AVAILABLE
```

Figure 3-4. Example 3270 Logon Window

On this logon window, you expect to find a string called *USERID*, i.e., this is the value for the **-expect** option. If this string is not found on this screen, then you have connected to the wrong program. If so, you would want to log off from this system and close your Term3270 connection. Therefore, a **catch** command is entered, which evaluates a **sendWait** operation.

The Term3270 method **sendWait** (Section 9.5.1.18) sends a 3270 function key to a system and waits for an expected return. In our example, **sendWait** expects to find the string *USERID* on the current screen. **catch** evaluates this **sendWait** operation. If **sendWait** doesn't find the expected string, the script executes a procedure that logs off from the system and then deletes the connection. If the string is not found, **catch** returns 1, therefore the **if** code is executed. If the expected string is found, the script continues executing, i.e., catch returns 0, therefore the **if** code is not executed and the script continues.

> **NOTE — syncImokLogoff** is a locally defined procedure used to log off from a system called IMOK.

3.3 Output Ownership Considerations

How you execute a script impacts ownership of any output files. When you submit a script for background execution, the system assigns ownership of the executing script and all output to the MYNAH Administrator or the person who started the MYNAH processes.

> **NOTE —** The following discussion assumes that the logid of the MYNAH Administrator is ***madmin*** and that ***madmin*** started the MYNAH processes.

> **NOTE —** This section uses the TermAsync method **sendWait** to illustrate these techniques. Briefly, **sendWait** sends a string, such as a login id or a UNIX command, to the connection and waits until an expected string is returned, such as
>
> ```
> sendWait "ls\r" -expect "$ "
> ```
>
> In this case, **sendWait** sends the **ls** command and waits until it receives the **$** prompt, which is the prompt of the initiator of the MYNAH processes. For a complete explanation of **sendWait**, see Section 8.4.1.11.

If you want your script to generate output files, we recommend the following measures.

3.3.1 Execution Directory Permissions When Using the BEE

When you execute a script using **xmyCmd submit**, the system generates several files in the directory containing the script. **madmin** must have permission to write these files in this directory. Typically, you will be part of the same UNIX group that **madmin** is. For example, if you decide to create a directory to contain all of your scripts, e.g. *scripts*, you should grant read/write permissions to **madmin** (and therefore the UNIX group to which you and **madmin** belong) as in

```
chmod 775 scripts
```

3.3.2 File Ownership When Using the BEE

Besides the system output files, you can have the MYNAH System save files generated by the script, such as a capture of the current screen or files transferred by an **ftp** session. If you execute the script from the background, ***madmin*** will own all of the files in the output directory and the output directory itself; you will have only read permissions for these files.

If this happens, you can include statements in your script to change the permissions of the user-specified output files.

The script in Figure 3-5 makes a connection, lists the contents and pathname of the current directory, and saves the most recent response and screen in the files */users/kjb/OUTPUT/screen* and */users/kjb/OUTPUT/path*, respectively. We then send a **chmod** command to change the permissions on those files.

```
xmyLoadPkg TermAsync
set conn1 [xmyTermAsync connect]
$conn1 sendWait “PS1=\”% \”\r”  -expect “% “
$conn1 wait -expect “% “
$conn1 sendWait “ls\r” -expect “% “
$conn1 sendWait “pwd\r” -expect “% “
$conn1 response -file /users/kjb/OUTPUT/path
$conn1 screen -file /users/kjb/OUTPUT/screen
$conn1 sendWait “chmod 666 /users/kjb/OUTPUT/path\r”\
                              -expect “% “
$conn1 sendWait “chmod 666 /users/kjb/OUTPUT/screen\r”\
                              -expect “% “
$conn1 disconnect
xmyExit
```

Figure 3-5. Example Script to Change Permissions of Output Files

3.4 Setting Output Levels

The amount of output that is generated is controlled in one of two ways:

- The MYNAH Administrator sets initial output levels by entering a value for the **OutputLevel** parameter of **Engine** entry in the *xmyConfig* file.
- You can control the output level for an entire script or part of a script using the **xmyVar(OutputLevel)** array variable. (See Section 5.12.)

Valid **OutputLevel** values for the events that can be recorded are

childscr	Records child script events.
compare	Records comparison events.
error	Records error events.
lang	Records language events.
script	Records script events.
summary	Records summary events.
sutimage	Records SUTimage events.
suttiming	Records SUT Timing events.
testobj	Records test object events.
user	Records user events.

Input to **xmyVar(OutputLevel)** is a Tcl list of zero or more of the **OutputLevel** event values listed above. In addition to these values, you can also use the following to set the output level:

- Setting **OutputLevel** to nothing using a empty string " "
- i.e.,

  ```
  set xmyVar(OutputLevel) " "
  ```

  means that no subsequent events will be recorded in the output file.

  > **NOTE —** The value for the **xmyVar(OutputLevel)** variable is returned or set using the Tcl command, **set**. See Section 4.1.4 and Appendix A.51 for information on using the **set** command.

- Setting **OutputLevel** to everything using an asterisk (*****)
- i.e.,

  ```
  set xmyVar(OutputLevel) *
  ```

  means that output will be created for all valid types.

It is suggested that:

- You should always choose to produce, at minimum, ***script*** and ***error*** events.
- You should never choose ***lang*** events for an entire script. These events are useful when attempting to debug a script but you should turn them on by using **xmyVar(OutputLevel)** with the ***lang*** value just before the lines of code that are causing the problem.

3.4.1 Returning the Current Output Level

To see what the current output level setting is, simply type

```
set xmyVar(OutputLevel)
```

In the following example, we see that script and error events are set.

```
> set xmyVar(OutputLevel)
script error
```

3.4.2 Changing the Output Level

To change the output level, enter the **OutputLevel** values for events you want to record as input to the **xmyVar(OutputLevel)** variable. The event values are entered as elements of a list delimited by braces ({}), for example

```
> set xmyVar(OutputLevel) {error childscr compare script user}
error childscr compare script user
```

The new output level will be ***exactly*** what you enter as the input to **xmyVar(OutputLevel)**, overwriting the previous level. For example, if your initial output level setting is the recommended minimum setting of producing ***script*** and ***error*** events and you want to temporarily produce output for ***lang*** events in addition to ***script*** and ***error*** events, you must enter all three event values as input to **xmyVar(OutputLevel)**, as in

```
> set xmyVar(OutputLevel) {script error lang}
script error lang
```

3.5 Script Termination

This section presents several hints to consider when you decide how to terminate your script.

3.5.1 Using a Termination Procedure

When a script terminates, there are common activities that should occur. For example, typical "end of script" processing involves producing a message in the script output file that summarizes script success or failure.

It is advisable to centralize all final processing in a termination procedure because:

— Scripts will be standardized: all scripts will perform the same "end of script" processing.

— Script maintenance will be reduced: any updates will be confined to one script.

Consider the example presented above: producing a standard message upon script termination. This could be done by checking the global variable *xmyVar(FailedCompares*) and passing an argument to the *xmyExit* command, as in the following example:

```
if {xmyVar(FailedCompares) == 0} {
        xmyExit "OK: no failed compares"
} else {
        xmyExit "ERROR: at least one compare failed"
}
```

Without a termination procedure, everyone who is coding scripts would have to include these commands in their scripts. Yet, with a termination procedure (e.g, **exitProc**), which would contain these same commands, the only command that's required in everyone’s scripts is the command to invoke the procedure, as in the following example:

```
exitProc
```

An alternative to including a command in the script to invoke the procedure would be to use the MYNAH termination procedure *ExitHandler*. This procedure can be invoked automatically when the script reaches its end by setting the global array variable *xmyVar(ExitHandler*). For more information about using this procedure, see Section 5.5.

3.5.2 Cleanup for Scripts Sent to ConnOnly and Fullstate SEs

For a script that is submitted to an SE whose state is *ConnOnly* or *Fullstate*, connections that were opened in the script remain active when the script terminates.

Because of this, the script should contain the code that will bring opened connections to an appropriate state in the system where they will be ready for use when the next script comes along for execution. For example, if the first script logs onto an application, the appropriate state for the next script could be the first screen after the *logon* process completes. That's where the first script should "leave off."

One method for coding for this situation is to include the required code in a procedure, possibly an *ExitHandler* procedure. For more information about using this procedure, see Section 5.5.

3.5.3 Sample Code

This section illustrates one simple example of the procedures used to terminate a script. The example assumes you have logged on to a system called ***imok*** and are testing a program called ***kurtz***.

Final script coding typically should follow a specific sequence. For example,

1. If logged onto a SUT, log off.
2. Disconnect the MYNAH connection.
3. Exit from the script.

> **NOTE —** This section contains excerpts from a script using the Term3270 Package. The commands and methods for the Term3270 Package are used to illustrate the script termination processes. For a description of the Term3270 Package, see Section 9.

Figure 3-6 shows an example of how the procedures used to terminate a script may be accessed by a script. The **KurtzOff** procedure (Figure 3-7) logs off from ***kurtz***, the **ImokLogoff** procedure (Figure 3-8) logs off from ***imok***, and the **GenEnd** procedure (Figure 3-9) exits the script.

```
# SCRIPT TERMINATION PROCEDURE CALLS AT BOTTOM OF SCRIPT
#if get here, need termination processing

#log off kurtz
KurtzOff $conn

#log off imok
ImokLogoff $conn

#exit the script
GenEnd $conn
```

Figure 3-6. Example Script Termination Procedures

Let's examine each of these procedures. Notice that each procedure has an argument for the connection handle. This "tells" the procedure the name of the connection in which to perform the commands.

Figure 3-7 shows the **KurtzOff** procedure. To exit from the ***kurtz*** system, you must enter the character "X" in row one, column two on the current screen. The procedure sends this character to the SUT via the **type** method. After pressing the enter key, it the ***Kurtz Selection Screen*** is not displayed, however, this will create an error, so the **catch** command is used to "catch the error." This allows you to code the script to manage the error condition. In this case, the procedure prints a message and returns a "0."

```
proc KurtzOff {conn} {

#LOG OFF KURTZ

    set expscreen {"KURTZ SELECTION SCREEN"}

    #access entity screen
    $conn type -text "X" -position {1 2}

    if {[catch {$conn sendWait -key enter -expect $expscreen}]} {
       xmyPrint "ERROR: did not get screen: $expscreen"
       return 0
    }

    #if get here, no errors
    return 1
}
```

Figure 3-7. Example SUT Log Off Script

Figure 3-8 shows the **ImokLogoff** procedure. To log off from the ***imok*** system, you must enter the string ***/rcl*** on the connection window. The **catch** command is used again (see preceding description).

```
proc ImokLogoff {conn} {
#PURPOSE: LOG OFF IMOK

    #log off
    $conn type -text "/rcl" -position {1 2}
    if {[catch {$conn sendWait -key enter}]} {
       xmyPrint "ERROR: /rcl failed"
       return 0
    }

    #if get here, no errors
    return 1
}
```

Figure 3-8. Example System Log Off Procedure

Figure 3-9 shows the **GenEnd** procedure. The procedure first uses the **global** command to make the **xmyVar** global array accessible to the procedure. This lets you print the counts generated by the **xmyVar** comparison elements. The script prints the contents of the **xmyVar** comparison elements. Finally, the script exits. If an exit string is specified, then the script exits with the exit string. This is very similar to what you might do if you are coding an ***ExitHandler*** procedure.

```
proc GenEnd {conn {exitstr ""}} {

# PURPOSE: TERMINATE A SCRIPT

    global xmyVar

    # PRINT COUNTS FROM GLOBAL ARRAY
    xmyPrint -text $xmyVar(GoodCompares)
    xmyPrint -text $xmyVar(FailedCompares)
    xmyPrint -text $xmyVar(WarningCompares)

    #finally, exit the script
    if {$exitstr != ""} {
       xmyExit $exitstr
    } else {
       xmyExit
    }
}
```

Figure 3-9. Example Script Exit Procedure

3.6 UNIX Commands in Scripts

You can include UNIX commands in your scripts, but how you use UNIX commands depends on what type of SE you use to execute the script.

If you submit a script using a Command-line SE, you can simply enter the UNIX command. If you are using an Embedded or Background SE, UNIX commands must be “exec”ed, that is you must preface the command with the Tcl **exec** command (Appendix A.13).

For example, in **xmytclsh** (for a Command-Line SE) typing

```
ping myserver
```

runs the UNIX **vi** command in the window **xmytclsh** is in. In the Background SE, a script containing the line **vi** ***/tmp/file*** will exit abnormally with the message

```
error: Auto execution of Unix commands only supported as
interactive commands. Use “exec” to execute “vi”
```

You must enter this command as

```
exec ping myserver
```

for Embedded or Background SEs.

You can use the **env** array to read and write $HOME environment variables. For example, to read in your home directory and assign it to a Tcl variable, you would type

```
set path $env(HOME)
/u/kjb
```

4. Tcl Basics

This section is designed to be a basic introduction to Tcl. After you finish this section, you should have a basic knowledge of the language. This section is divided into several subsections, each detailing a specific Tcl category. Table 4-1 lists these subsections.

Table 4-1. Section Contents

Section Name	Description	Section Number
Before We Begin	This subsection introduces some basic ideas and concepts, including a discussion of some basic Tcl commands.	4.1, Page 4–2
Expressions	This subsection discusses the basics of Tcl expressions using the TCL operands, operators, and math functions.	4.2, Page 4–8
Tcl Syntax	This subsection describes the Tcl syntax, the rules that determine how commands are used, focusing on substitution, quoting, and commenting.	4.3, Page 4–16
Lists and Arrays	This subsection describes the Tcl facilities for handling collections of data.	4.4, Page 4–22
Control Flows	This subsection describes the Tcl commands (e.g. **if**, **while**, and **for**) for controlling the flow of script execution.	4.5, Page 4–37
Tcl Error/Exception Procedures	This subsection describes Tcl command for trapping errors so that a script does not abort.	4.6, Page 4–47
Procedures	This subsection describes the methods of creating new Tcl command procedures.	4.7, Page 4–51
String Manipulation	This subsection describes the basics of pattern matching for string manipulation.	4.8, Page 4–54
File Input/Output	This subsection describes basic Tcl input/output functions, concentrating on opening and closing files.	4.9, Page 4–56
Using xmytclsh	This subsection describes the use of the **xmytclsh** program to test Tcl commands.	4.10, Page 4–60
Importing Scripts Using the source Command	This subsection provides basic information on how to import scripts for execution.	4.11, Page 4–61

4.1 Before We Begin

There are some basic concepts and ideas that we must first mention before we begin. In addition, a rudimentary knowledge of a few basic Tcl commands, **set**, **unset**, **expr**, **append** and **incr**, which are used in examples in the following sections, will also be helpful.

> **NOTE —** These are only brief descriptions of these basic commands. Complete descriptions of these commands and all basic Tcl commands can be found in Appendix A.

4.1.1 Using the TclX Help Facility

TclX provides a help facility, **tclhelp**. During installation, the MYNAH System places **tclhelp** in *$XMYDIR/contrib/tclhelp/bin*. To use this facility, type, at the UNIX prompt,

```
$XMYDIR/contrib/tclhelp/bin/tclhelp &
```

The window in Figure 4-1 appears.

Figure 4-1. tclhelp Window

The help is organized in a tree structure with each subject button opening a different branch. For example, if you click on the **tcl/** button, the window in Figure 4-2 appears. The subjects in this window are organized according to the various TclX extension categories listed in Appendix B.1.

Figure 4-2. tclhelp Tcl Subjects Window

See Appendix B.14 or click on the **Help** button in Figure 4-1 for information on using the help facility.

4.1.2 Examples in this Document

The MYNAH System provides a utility program, **xmytclsh** (Section 4.10), that lets you interactively run Tcl commands, including all MYNAH extensions.

When you start the program, the **xmytclsh** prompt (>) appears. The response returned from each command entered to **xmytclsh** appears on a separate line without the prompt, as in;

```
xmytclsh
> set x 5
5
```

The examples in this document have been created as if they were entered using **xmytclsh**. You, in turn, may wish to use **xmytclsh** to verify the examples included in this section or to test your own examples.

4.1.3 Basic Concepts and Definitions

The only data type in Tcl is a **string**. All commands, arguments, and results are strings. A Tcl string, however, can represent, among other things, integers, floating point numbers, and lists. A string containing words separated by white space is a **list**.

Tcl commands are similar in form to shell commands: each command consists of one or more words separated by spaces or tabs, and <newline> or a semi-colon (;) ends a command. The first word of each command is always the command name and the rest of the words are interpreted as arguments. What forms the arguments take are determined by each individual command.

One of the basic Tcl strings is the **variable**, a user defined quantity that can assume one of a set of values. You can use the variable to act as input to Tcl commands.

The following subsections describe the basic Tcl commands used to create and manipulate strings. Since they are so basic, we concentrate on creating and manipulating variables.

4.1.4 set

Syntax

```
set variable ?value?
```

Return

Value of the variable

Description

The **set** command assigns or “sets” a string. **set** takes one or two arguments.

When using two arguments, **set** assigns the value to the variable. The value can be an integer, string, or expression.

When using one argument, **set** returns the current value of the variable.

Example

```
> set a 2
2
> set b hello
hello
> set a
2
```

4.1.5 unset

Syntax

`unset` *variable ?variable? …*

Return

No return

Description

The **unset** command destroys a variable created by **set**. **unset** takes at least one argument.

Example

To remove both of the variables we created in the previous section, type

```
unset a b
```

4.1.6 expr

Syntax

`expr` *argument ?argument? …*

Return

String of the expression's value

Description

The **expr** command evaluates an **expression**, which is a combination of values (called operands) and operators. Specifically, **expr** concatenates an argument, evaluates the result as a Tcl expression, and returns the value.

Example

```
> expr 6.7 + 5.5
12.2
```

4.1.7 incr

Syntax

`incr` *`variable ?value?`*

Return

New value of the variable incremented by the specified value

Description

The **incr** command increments a variable by the supplied value. **incr** takes either one or two arguments, a required variable and an optional incremental value. The incremental value must be a positive or negative integer. If you don't specify a value, Tcl automatically adds one (1) to the variable.

Example

```
> set x 12
> incr x 5
17
> incr x
18
> incr x -7
11
```

4.1.8 append

Syntax

`append` *`variable value ?value?`* …

Return

Appended value of the variable

Description

The **append** command appends one or more values to a variable, that is they are added to the end of the variable. **append** takes two or more arguments.

The appended values are treated as text. Instead of adding the two integers together as **incr** does, **append** simply treated them as if they were letters.

Example

```
> set x 5
5
> append x 6
56
> append x Y
56Y
```

4.1.9 history

Syntax

history ?*option*? ?*arg arg ...*?

Return

A listing of previously executed commands

Description

The **history** command lets you perform several actions on **events**, which are previously executed commands. In it's simpliest form, **history** returns a **history list**, a list of these events with an event number assigned to each event.

> **NOTE —** By default, only the 20 most recent events are returned, but you can tell history to retain more events.

history takes several options that let you perform other functions besides returning the history list, including re-executing and changing a command.

Example

As we said, the simpliest use of **history** is to return the history list, as in the following:

```
> xmyLoadPkg TermAsync
> set x 5
5
> set y 7
7
> history
     1  xmyLoadPkg TermAsync
     2  set x 5
     3  set y 7
     4  history
```

In the following examples, the **redo** option re-executes the second command from the above list, the **change** option replaces the value of the original second command, and the **event** option returns the new value of the second command.

```
> history redo 2
5
> history change "set x 7.7" 2
> history event 2
set x 7.7
```

Lastly, **history** is re-executed to confirm the change and **redo** generates the new value of the variable **x**.

```
> history
     1  xmyLoadPkg TermAsync
     2  set x 7.7
     3  set y 7
     4  history
     5  set x 5
     6  history change "set x 7.7" 2
     7  history event 2
     8  history
> history redo 2
7.7
```

4.2 Expressions

As we mentioned in Section 4.1.6, expressions are a combination of values (called operands) and operators. All Tcl expression commands require one or more arguments. **expr** is only one of these commands and is in fact the simplest. For example, **if** requires two arguments, both of which can be expressions: the result from the evaluation of the first is used to determine whether or not to evaluate the second. (See Section 4.5.1 for a discussion of the **if** command.)

4.2.1 Operands

Normally, operands are integers or real numbers, and these can be variables or constants. As we've mentioned, variables are set using the **set** command.

Constants are usually decimal (base 10). There are, however, ways to use different number bases. If the first character is a 0 (zero), then the constant is considered to be octal (base 8). If the first two characters are 0x, then the number is considered to be hexadecimal (base 16). For instance, the examples in Table 4-2 are all equal to the decimal constant 422.

Table 4-2. Tcl Number Base Example

Constant	Base
422	decimal
0646	octal
0x1A6	hexadecimal

For floating-point numbers, Tcl accepts most of the forms defined for ANSI C with the exception of the f, F, l, and L suffixes. Figure 4-3 shows valid examples of floating-point numbers.

```
5.3
6.28e+7
7E5
9.
```

Figure 4-3. Floating-Point Number Examples

4.2.2 Operators

Tcl supports arithmetic, relational, logical, bitwise, and choice operators similar to those used in ANSI C expressions. As with ANSI C, Tcl treats a zero value (such as for a relational or logical operator) as false and anything else as true. Tcl uses 1 for true and 0 for false.

In each of the following sections, the operators are listed in the order of precedence. Additionally, the types of operators are listed in the general order of precedence. The exact order of precedence is discussed in Section 4.2.2.6.

4.2.2.1 Arithmetic Operators

Tcl supports the arithmetic operators -, +, *, /, and %. The - operator can be used for two operations:

- For negation, as in *-(7*$x)*
- For subtraction, as in *7-5*

Table 4-3 lists the Tcl arithmetic operators.

Table 4-3. Tcl Arithmetic Operators

Syntax	Results
-x	Return the negative of *x*.
x*y	Multiply *x* and *y*.
x/y	Divide *x* by *y*. If both operands are integers, the result is truncated to an integer.
x%y	Remainder when dividing *x* by *y*. Both operands must be integers. This is called the modulus operator.
x+y	Add *x* and *y*.
x-y	Subtract *y* from *x*.

4.2.2.2 Relational Operators

Tcl supports the relational operators <, >, <=, and >=, which are used to compare two values. If the operands meet the conditions, the operators produce a true (1) result. If the operands do not meet the conditions, the operators produce a false (0) result. Table 4-3 lists the Tcl arithmetic operators.

Table 4-4. Tcl Relational Operators

Syntax	Description	Results
x<y	Less than	1 if *x* is less than *y*, else 0
x>y	Greater than	1 if *x* is greater than *y*, else 0
x<=y	Less than or equal	1 if *x* is less than or equal to *y*, else 0
x>=y	Greater than or equal	1 if *x* is greater than or equal to *y*, else 0
x==y	Equal to	1 if *x* is equal to *y*, else 0
x!=y	Not equal to	1 if *x* is not equal to *y*, else 0

4.2.2.3 Logical Operators

Tcl supports the logical operators *!*, *&&*, and ||, which are used to combine the results of relational operators, such as

```
($x > 6) || ($x < 13)
```

Table 4-3 lists the logical Tcl operators.

Table 4-5. Tcl Logical Operators

Syntax	Description	Results
!x	Logical NOT of *x*.	1 if *x* is a zero, else 0.
x&&y	Logical AND of *x* and *y*.	1 if *x* and *y* are non-zero, else 0.
x||y	Logical OR of *x* and *y*.	1 if either *x* or *y* is non-zero, else 0.

4.2.2.4 Bitwise Operators

Tcl supports the logical operators ~, <<, >>, &, ^, and |, which manipulate the individual bits of integers. Table 4-3 lists the Tcl bitwise operators.

Table 4-6. Tcl Bitwise Operators

Syntax	Description	Results
~x	Bitwise complement (ones complement) of *x*.	The result is bits that are the opposite of those in the operand: 1's replace 0's and 0's replace 1's.
x<<y	Left-shift *x* by *y* bits.	0's are shifted into the low-order bits.
x>>y	Arithmetic right-shift *x* by *y* bits.	0's are shifted in for positive numbers and 1's are shifted in for negative number.
x&y	Bitwise AND of *x* and *y*.	Each bit of the result is generated by applying the AND function to the corresponding bits of the *x* and *y* operands.
x^y	Bitwise exclusive OR of *x* and *y*.	Each bit of the result is generated by applying the exclusive OR function to the corresponding bits of the *x* and *y* operands.
x|y	Bitwise OR of *x* and *y*.	Each bit of the result is generated by applying the OR function to the corresponding bits of the *x* and *y* operands.

4.2.2.5 Choice Operator

Tcl has only one choice operator, **?**, which lets you select one of two results. The choice operator takes three operands and has the syntax.

```
x?y:z
```

If **x** is true, then **y**. If **x** is false, then **z**.

The first operand is evaluated. If the solution is true (non-zero), then the operand following the ? is evaluated and becomes the result of the expression; if the solution is zero (false), the second operand is evaluated and becomes the result.

4.2.2.6 Precedence

Table 4-7 lists all of the Tcl-supported operators described in the previous subsections; the horizontal lines separate the operators into groups with the same precedence. This table also lists the operand type (integer, floating-point, and string) for each operator.

Table 4-7. Tcl Operator Precedence

Operator	Operand Types
-x	integer, floating-point
!x	integer, floating-point
~x	integer
x*y	integer, floating-point
x/y	integer, floating-point
x%y	integer
x+y	integer, floating-point
x-y	integer, floating-point
x<<y	integer
x>>y	integer
x<y	integer, floating-point, string
x>y	integer, floating-point, string
x<=y	integer, floating-point, string
x>=y	integer, floating-point, string
x==y	integer, floating-point, string
x!=y	integer, floating-point, string
x&y	integer
x^y	integer
x|y	integer
x&&y	integer, floating-point
x||y	integer, floating-point
x?y:z	*x*: integer, floating-point

Operators with higher precedence appear above operators with lower precedence. For example,

```
expr (11/3<=4)
```

yields the result true (1) since the / operator has a higher precedence than the <= operator.

You can use parenthesis to indicate how you want operators to be evaluated; whatever is in the parenthesis will be evaluated first. For example,

```
expr (7*(5+4))
```

yields 63, but

```
expr (7*5+4)
```

yields 39.

Operators with the same precedence level are grouped from left to right. For example *7+5-3* and *(7+5)-3* are equal; both evaluate to 9.

4.2.3 Mathematical Functions

Tcl supports many mathematical functions, such as *sin* (the trigonometric [trig] sine function) and *sqrt* (square root). These functions can be used as arguments for expressions and are invoked using standard functional notation, as in the following examples:

```
expr 3*tan($a)
expr sqrt($x) + $b
```

Table 4-8 lists the Tcl-supported trigonometric functions.

NOTE — All arguments for trigonometric functions are expressed in radians.

Table 4-8. Tcl Trig Functions

Function	Description
acos(x)	Arc cosine of *x*, in the range $0 \le x \le \pi$.
asin(x)	Arc sine of *x*, in the range $-\pi/2 \le x \le \pi/2$.
atan(x)	Arc tangent of *x*, in the range $-\pi/2 \le x \le \pi/2$.
atan2(x,y)	Arc tangent of *x/y*, in the range $-\pi/2 \le x \le \pi/2$.
cos(x)	Cosine of *x*.
cosh(x)	Hyperbolic cosine of *x*.
sin(x)	Sine of *x*.
sinh(x)	Hyperbolic sine of *x*.
tan(x)	Tangent of *x*.
tanh(x)	Hyperbolic tangent of *x*.

Table 4-9 lists the standard Tcl-supported math functions.

Table 4-9. Tcl Math Functions

Function	Description
abs(x)	Absolute value of *x*.
ceil(x)	Smallest integer not less than *x*.
double(x)	Real value equal to the integer *x*.
exp(x)	*e* raised to the *x*th power, i.e., e^x.
floor(x)	Largest integer not greater than *x*.
fmod(x,y)	Floating-point remainder of *x/y*.
hypot(x)	Square root of ($x^2 + y^2$). This is the hypotenuse of a right triangle.
int(x)	Integer from truncating *x*.
log(x)	Natural logarithm of *x*.
log10(x)	Base 10 logarithm of *x*.
pow(x,y)	*x* raised to the *y*th power, i.e., x^y.
round(x)	Integer from rounding *x*.
sqrt(x)	Square root of *x*.

4.2.4 Conversion

When you use operands of different types, Tcl converts them to the same type, e.g., if one in an integer and the other is floating-point, both are converted to floating-point.

4.3 Tcl Syntax

This section describes the Tcl syntax, the rules that determine how commands are used.

4.3.1 Substitution

Tcl provides three forms of substitution: variable, command, and backslash. Each substitution causes some of the original characters of a word to be replaced with another value.

4.3.1.1 Variable

Variable substitution using *"$"* lets you use the value of a variable in a command; characters following the $ are treated as a variable. A basic example would be to type

```
set c $a
```

The *$* causes the Tcl interpreter to perform the substitution, assigning the current value of the variable *a* to the variable *c*.

A more illustrative example would be

```
> set a 6
> expr $a+3
9
```

where the variable *a* is assigned the value 6, which, using the **expr** command, is added to 3.

You can have variable substitutions anywhere in an argument, such as

```
> set a 6
> expr $a*$a
36
```

In this example, two substitutions occurred within the same argument, i.e., 6 times 6.

4.3.1.2 Command

Tcl also lets you use the result of a command as an argument to another command. For this, you put square brackets around a command or script excerpt. For example, type

```
> set a 6
> set c [expr $a/2]
3
```

Everything inside the square brackets is evaluated by Tcl as a separate command. In this case, the command happens to be **expr**. The value of *a* is divided by 2 and this new value is assigned to *c*. Square brackets can be nested. Try entering

```
set e [expr $c*[expr $a+4]]
```

Given the values of *a* and *c* from the previous example (6 and 3, respectively), *e* would assume the value 30 (i.e., *4*(6+4)*).

4.3.1.3 Backslash

The backslash character overrides the special meaning of the following character, such as a *$*, a space, or a left bracket ([); the character is interpreted literally and not according to its Tcl-based characteristics. If you typed

```
> set d \$a
$a
```

the variable *d* is literally set to the characters $a and not 6, assuming the value for *a* from the previous examples.

Tcl does not allow embedded spaces in words, but the backslash lets you use spaces in arguments, such as

```
> set x this\ string\ uses\ embedded\ spaces
this string uses embedded spaces
```

A backslash suppresses only the Tcl interpretation of the character immediately following it. In this example

```
> set x 6.5; set y \$$x
$6.5
```

the first dollar sign is passed as a literal character while the second is used for variable substitution.

The backslash can also be used to insert special characters and functions, including all sequences defined for ANSI C, such as *t* for a tab. Table 4-10 lists all Tcl-supported backslash sequences.

Table 4-10. Tcl Backslash Sequences

Backslash Sequence	Replaced By
\a	Audible alert (0x7)
\b	Backspace (0x8)
\f	Form feed (0xc)
\n	Newline (0xa)
\r	Carriage return (0xd)
\t	Tab (0x9)
\v	Vertical tab (0xb)
\ddd	Octal value
\xhh	Hex value from hh
\{space}	A single space character

4.3.2 Quoting

Much like the backslash character, quoting suppresses the effect of special characters (white spaces, new-lines, semicolons). While only the character following a backslash is suppressed, everything quoted is suppressed, such as when you want an argument to include embedded spaces.

Tcl lets you use two forms of quoting: double quotes and braces. The quote character must be the first character of the quoted argument. The quote character is stripped off when the argument is passed to the interpreter.

4.3.2.1 Using Double Quotes

When using double quotes, everything from the first double quote to the closing double quote is interpreted as one command argument; all spaces, tabs, newlines, and semicolons in the argument lose their special meaning. Dollar signs and brackets are not affected by double quotes. You can retype the second example in Section 4.3.1.3 without using a backslash in the following way.

```
> set x "this string uses embedded spaces"
this string uses embedded spaces
```

Similarly, you could type

```
> set d "$a"
6
```

assuming *$a* is still set to the value from Section 4.3.1.2.

A slightly more elaborate example is

```
> set x 3.1
> set test "x is $x\n$x squared is [expr $x*$x]"
x is 3.1
3.1 squared is 9.61
```

NOTE — This can also be written

```
set x 3.1
set test "x is $x
$x squared is [expr $x*$x]"
```

with the newline replacing the newline character (*n*).

4.3.2.2 Using Braces

You can also use the braces to suppress special characters. While double quotes disable only word and command separators (e.g., white spaces, newlines, semi-colons), braces disable almost all special characters. For example, the last example for double quotes rewritten using braces would yield the following:

```
> set x 3.1
> set test {x is $x\n$x squared is [expr $x*$x]}
x is $x\n$x squared is [expr $x*$x]
```

Because the braces suppress the variable substitution for the $, the characters *$x\n$x* and $x*$x are interpreted literally.

> **NOTE —** The next example uses the **while** command. Briefly, the syntax for **while** is
>
> `while` *`test while_body`*
>
> **while** evaluates *test*. If the result is zero, execution of the **while** loop is terminated and the next command in the Tcl script is executed. If the result is non-zero, **while** executes the *while_body* script, re-evaluates *test*, and continues until the *test* result is zero. (See Section 4.5.2.1 for a further discussion on the **while** command.)

A primary use of braces is to defer evaluation. Normally, the Tcl parser immediately processes special characters. Using deferred evaluation, special characters are passed as part of the argument to a command procedure, which then processes the characters. Consider the following script that calculates the square of the first five integers.

```
> set x 1
> while {$x < 5} {
      set square [expr $x*$x]
      set x [expr $x+1]
}
```

> **NOTE —** When using braces, the **xmytclsh** prompt does not appear until you enter the closing brace.

The braces enclose the **`test`** and **`while_body`** parts to defer substitution. The **`test`** expression *$x<5* is evaluated at the beginning of each iteration and is used to determine whether or not to terminate the **while** loop. This way, the substitution is done anew each time **while** evaluates the script. If you had typed

```
while $x<5
```

the **`test`** expression would have been evaluated once as a constant expression, i.e., *1<5*, and the loop would have executed forever.

4.3.3 Comments

You can add comments to a script by entering a pound sign (#) as the first non-blank character of a line; all characters following the # until a newline will be treated as a comment. If the # appears anywhere other then as the first non-blank character of a line, it will be treated as an ordinary character unless it is immediately preceded by a semicolon (;).

```
# This is a correct comment
> set x 3  # This is not a comment
> set x 3; # This is a correct comment
```

In the first example, the # appears in the first non-blank character, so this is a valid comment. In the second example, the # appears in the middle of the command and is considered an argument for the **set** command. (This would generate an error since the **set** command now has too many arguments.) In the last example, the semicolon (;) terminates the **set** command, and the # is therefore the first non-blank character following the semicolon.

4.4 Lists and Arrays

As we said in Section 4.1.3, the only Tcl data type is the string. However, Tcl also provides facilities for handling collections of such data.

A Tcl list is an ordered collection of components of a string, which are called **elements**. Syntactically, it is just a string containing one or more fields (such as words, integers, and file names) separated by white spaces or tabs.

> **NOTE —** When working with a list, an index of 0 refers to the first element of the list, 1 refers to the second element, etc. An empty string is generated when the index is outside the range of the list.

An **array** is similar to a list in that both are collections of elements. However, where a list is a string of elements, an array's elements are variables, each with its own name and value. Each variable can only be referenced with the array name.

4.4.1 Creating Lists

As we said, a Tcl list is an ordered collection of string elements.

The simplest way to create a list is to use the **set** command. This is fine if you're creating simple lists. To create more complex lists, Tcl provides two commands, **concat** and **list**. These two commands are similar, but they differ in the how they combine the arguments into a list.

4.4.1.1 Using the set Command

The simplest way to create a list is to use the **set** command:

```
> set x {a b c d}
a b c d
```

This creates a list with four elements, a, b, c, and d.

Usually, you will use braces to enclose the elements of a list; this passes the entire list to the command as a single argument. Braces let you enclose list element to permit spaces. Backslashes let you enter special characters in a list. You can also use braces to nest lists within lists:

```
set x {a b {c d e} f { }}
```

In this case, you created a list with five elements:

- The character a
- The character b
- A sublist containing the characters c, d, and e
- The character f
- An empty list.

> **NOTE —** When using braces to enclose elements of a list, the braces will not appear unless you use backslash substitution to insert braces (or other special characters) into the list.

4.4.1.2 concat

Syntax

`concat` *value_1 value_2 ?value_3? …*

Return

Concatenated list

Description

The **concat** command takes any number of arguments and creates one large list. Each argument is considered a list, whether it is an individual element or an existing list. If you do not supply any arguments, **concat** returns an empty string.

Example

```
> concat {a b c} d {e f} g h
a b c d e f g h
```

Since each argument is considered a list, if an argument contains a list, the list becomes an individual element of the new list

```
> concat {a {b c}} d {e f} g h
a {b c} d e f g h
```

You can use double-quotes and **set** to achieve the same affect.

```
> set x {1 2 3}
> set y {4 5}
> set z {7 8 9}
> set w [concat $x $y $z]
1 2 3 4 5 6 7 8 9
> set w "$x $y $z"
1 2 3 4 5 6 7 8 9
```

4.4.1.3 list

Syntax

`list` *value_1 value_2 ?value_3? …*

Return

Concatenated list

Description

The **list** command takes any number of arguments and creates one large list. Unlike **concat**, **list** retains the integrity of elements in the argument so that they remain discrete elements in the resulting list. If you do not supply any arguments, **list** returns an empty string.

Example

```
> list {a b c} d {e f} g h
{a b c} d {e f} g h
```

The resulting list contains five elements, compared to **concat**, which would create a list with eight elements.

```
> concat {a b c} d {e f} g h
a b c d e f g h
```

You can use **concat** and **list** as part of the argument for another expression, such as using **set** to assign a list to a variable:

```
> set x [concat {a b c} d {e f} g h]
a b c d e f g h
> set y [list {a b c} d {e f} g h]
{a b c} d {e f} g h
```

4.4.1.4 llength

Syntax

`llength` *list*

Return

Number of elements in a list

Description

The **llength** command calculates the number of elements in a list. **llength** takes only one argument, the list itself.

Example

```
> llength {{a b c} d {e f} g h}
5
> llength {a b {c d e}}
3
> llength a
1
> llength { }
0
```

4.4.2 Extracting Elements from a List - lindex

Syntax

`lindex` *`list index`*

Return

List element

Description

The **lindex** command extracts elements from a list. Each element corresponds to an entered index. The list index begins at 0.

Example

To extract the second element in a list, type

```
> lindex {a b c d} 1
b
```

To extract the fourth element in a list, type

```
> lindex {a b {c d e} f} 3
f
```

4.4.3 Modifying Lists

Tcl provides several commands that let you modify lists.

4.4.3.1 lappend

Syntax

`lappend` *`variable value ?value? …`*

Return

An appended list

Description

The **lappend** command lets you append new elements to a list assigned to a variable. Each additional argument is appended to the end of the variable's list as a new list element.

lappend enforces proper list structure and is implemented so as to avoid string copies, which helps in performance when working with large lists. This can simplify creating lists.

Example

```
> set x {a {b c d} e}
a {b c d} e
> lappend x X {Y Z} WW
{a {b c d} e} X {Y Z} WW
```

Both of the following would create the same list, but using **lappend** is more efficient if one of the lists is very large.

```
> lappend x $a $b $c
> set x "$x [list $a $b $c]"
```

4.4.3.2 linsert

Syntax

`linsert` *`list index value ?value? …`*

Return

A list containing inserted elements

Description

The **linsert** command lets you form a new list by inserting new elements into an existing list.

linsert generates a new list by inserting the new elements just before the element specified by the index.

Example

```
> set x {{a b c} d {e f}}
{a b c} d {e f}

> linsert $x 1 X {Y Z}
{a b c} X {Y Z} d {e f}

> linsert $x 0 X {Y Z}
X {Y Z} {a b c} d {e f}

> linsert $x 5 X {Y Z}
{a b c} d {e f} X {Y Z}
```

In the first example, the index is 1, so the new elements were inserted after the first element in the old list. In the second example, the new elements were inserted at the beginning of the old list. In the third example, the index was greater than the number of elements in the original list, so the new elements were inserted at the end of the list.

4.4.3.3 lreplace

Syntax

`lreplace` *`list index_1 index_2 ?value? ?value? …`*

Return

A list containing deleted or replaced elements

Description

The **lreplace** command lets you delete elements from a list and lets you optionally add new elements in their place.

The *index_1* and *index_2* arguments are, respectively, the indices of the first and last elements to be deleted. If you specify any additional arguments, they are inserted in place of the elements you deleted.

Example

In this first example, only three arguments are specified, and the third and fourth elements are deleted from a list and the new list assigned to the variable *x*.

```
> set x [lreplace {a {b c} {d e f} g h {i j} k} 2 3]
a {b c} h {i j} k
```

In this second example, we take the list assigned to the variable *x* in the previous example, replace the third, fourth, and fifth elements with new elements, and assigned this list to the variable *y*.

```
> set y [lreplace $x 2 4 W {X Y} Z]
a {b c} W {X Y} Z
```

4.4.3.4 lrange

Syntax

`lrange` *`list index_1 index_2`*

Return

List containing extracted elements

Description

The **lrange** command lets you extract a range of elements from a list.

The resulting list consists of the elements that lie inclusively between the two indices.

Example

Let's take the list assigned from the last example. In the first example, we simply extract and display the third, fourth, and fifth elements from the list. In the second example, we extract the first and second elements.

```
> lrange $y 2 4
W {X Y} Z
> set z [lrange $y 0 1]
a {b c}
```

4.4.4 Searching Lists - lsearch

Syntax

```
lsearch ?mode? list pattern
```

Return

Index number

Description

The **lsearch** command searches a list for an element with a particular value.

lsearch returns the index of the first element in the list that matches the pattern, or -1 if no element matched the pattern.

The three optional mode arguments, if used, appear before the list argument and determine how the elements of the list are matched against the pattern:

-exact	The list element must exactly match the pattern.
-glob	Each list element is matched against a glob-style pattern that follows the rules for the **string match** command as described in Appendix A.54. This is the default mode.
-regexp	Each list element is matched against a regular expression pattern that follows the rules for the **regexp** command as described in Appendix A.45.

Example

```
> set x {Prufrock Ken Regis Salinger}
> lsearch $x Prufrock
0

> lsearch $x S
-1

> lsearch $x S*
3

> lsearch $x Updike
-1

> lsearch -exact $x Regis
2
```

4.4.5 Sorting Lists

Syntax

```
lsort ?options? list
```

Return

Sorted list

Description

The **lsort** command sorts a list. **lsort** takes one or more arguments.

By default, **lsort** sorts a list in increasing lexicographical order, that is numbers are treated as letters and the list is sorted alphabetically. **lsort** looks at the relative position of each character of each element and sorts accordingly.

lsort support several options that let you control how and what to sort, including the two following examples:

-decreasing Sort the list in decreasing order.

```
> lsort -decreasing {Prufrock Ken Klaatu Salinger}
Salinger Prufrock Klaatu Ken
```

-integer The elements are treated as integers and sorted according to integer value:

```
> lsort -integer {12 1 3 103 9}
1 3 9 12 103
```

You can specify your own sorting functions using the **-command** option. See the **lsort** manpage on page A–57 for an explanation of the **-command** option.

Example

```
> lsort {Prufrock Ken Klaatu Salinger}
Ken Klaatu Prufrock Salinger
> lsort {12 1 3 103 9}
1 103 12 3 9
```

In both examples, **lsort** first sorted the elements according to the first character of each element, then by the second character, and so on. Therefore, rather than returning

```
1 12 103 3 9
```

for the second example, which we could expect when sorting numbers, **lsort** first sorted by the first character, found three elements beginning with 1, and then sorted these three elements according the second character of each element.

4.4.6 Converting Between Strings and Lists

Tcl provides the **split** and **join** commands for converting lists to strings and strings to lists.

4.4.6.1 split

Description

`split` *`string ?split_character? …`*

Return

A list created from a string.

Description

split breaks up the pieces of the string into elements of a list at the *split characters*. There can be one or more split characters. This lets you process each element of the list independently.

If the split character appears at the beginning of a string, **split** generates an empty element. Empty elements would also be generated if there were consecutive split characters or if the split characters appeared at the end of the string.

You can specify several split characters; the order of the split characters doesn't matter.

If you specify an empty string as the split characters, each character of the string becomes a separate list element separated by an empty element.

Example

To work on each element of a date or UNIX pathname, both of which are usually separated by slashes, type:

```
> set x 5/16/95
> set y /usr/bin/pwd
> split $x /
5 16 95
> split $y /
{} usr bin pwd
> split "Tom Dick Harry" {}
T o m { } D i c k { } H a r r y
```

Remember, the order of the split characters doesn't matter:

```
> split 12xy356y7 xy
12 { } 356 7
```

One of each of the split characters appeared between 12 and 356, so **split** generated an empty element.

4.4.6.2 join

Syntax

```
join list separator
```

Return

A string created from a list.

Example

The **join** command creates a string from list elements. This is the opposite function of **split**. **join** concatenates the list elements with the separator string between each pair of elements. You can enter any number of characters for the separator string. If the list contains an empty element, the separator string replaces the empty element.

You can use an operator as the separator string. You can then generate a Tcl expression when you join the list elements.

Example

```
> join { {} usr bin pwd} /
/usr/bin/pwd

> set x {4 9 25}
4 9 25
> set y [sqrt([join $x *]]
30
```

4.4.7 Arrays

An array is also a collection of elements, but an array's elements are variables, each with its own name and value. Each variable can only be referenced with the array name by using the syntax

arrayname*(elementname)*

A Tcl array is usually known as an "associative" array, since its element names can be any arbitrary strings.

For example, to assign a value to the element *charlie* in the array *pet*, type

```
> set pet(charlie) snoopy
```

To add two more elements into the array, you can do the same

```
> set pet(alice) mimi
> set pet(tom) tequila
```

Tcl provides a built-in command, **array**, for obtaining information on an array. Now type

```
> array size pet
3
```

array command takes as its first argument a keyword, indicating the kind of information requested. Subsequent arguments would depend on the keyword. In this example, we queried the size of the array *pet*.

Another often used query is *names*, which returns a list containing the names of all the elements in the array. Try typing

```
> array names pet
tom charlie alice
```

4.5 Control Flows

The Tcl commands for controlling the execution flow of a script are similar to those used by ANSI C and `csh`.

4.5.1 if

Syntax

`if` *`expr body`*

`if` *`expr1`* `then` *`body1`* `elseif` *`expr2`* `then` *`body2`* … `else` *`bodyn`*

Return

Result of conditionally executed body

Description

The most basic Tcl control command is **if**, which in its simplest form takes only two arguments expr (B_____) and body (a block of Tcl code). When expr evaluates true (non-zero) body is executed, otherwise control passes to the Tcl statement following body

```
> set x -5
-5
> if {$x < 0 } {
      set x 0
}
0
```

If the first line had been

```
set x 5
```

the expression *$x < 0* would have been false and body ({ set x 0 }) would not be executed. After you will want to execute different code when expr evaluates false and will use the second form of the if statement.

```
> set x 5
5
> if {$x < 0 } {
        set x 0 } else {
        set x [expr $x+1]
}
6
```

Using the **elseif** keyword in the second form of it you can create chained conditionals as in the next example. **If** checks to see if the variable *x* is less than zero. If it isn't, the first **elseif** clause checks to see if *x* is equal to zero, and the second checks to see if *x* is greater than zero.

> **NOTE —** The **then** and **else** arguments are considered optional "noise words." A primary function is to help in readability. **if** will execute without them. However, if you do not include an **else** clause and no tests succeed, **if** will return an empty string.

Example

```
> set x 5
> if {$x < 0 } then {
            set x [expr $x + 1]
}    elseif {$x == 0} {
            echo x equals zero
}    elseif {$x > 0} {
            set x [expr $x - 1]
}
4
```

This example is equivalent to the prior example except for the omission of the second **elseif** clause, which is redundant.

```
> set x 5
> if {$x < 0 } then {
            set x [expr $x + 1]
}    elseif {$x == 0} {
            echo x equals zero
}    else {
            set x [expr $x - 1]
}
4
```

4.5.2 Looping Commands

You can use the **while**, **for**, and **foreach** commands for looping, executing nested scripts over and over. **while** and **for** are similar to the corresponding C statements; **foreach** is similar to the corresponding feature of the csh shell. The setup and termination criteria are different for each command.

4.5.2.1 while

Syntax

`while` *`test body`*

Return

Result of conditionally executed body

Description

while is similar to **if** in that it evaluates a test expression and executes a body if the result is true (non-zero). Unlike **if**, **while** repeats the process until the expression evaluates to zero, at which point **while** terminates and returns an empty string.

Example

The following script creates a list containing, in decreasing order, the squares of the first five non-zero integers:

```
> set x 5
> set y ""
> while {$x > 0} {
      lappend y [expr $x * $x]
      incr x -1
}
> set y
25 16 9 4 1
```

In the body you must include code that modifies the test expression, otherwise it will never evaluate to zero. This reinitialization script is evaluated after the loop body is executed. If we start with *x* set to a negative number, we would rewrite the script like this:

```
> set x -5
> set y ""
> while {$x < 0} {
      lappend y [expr $x * $x]
      incr x
}
> set y
25 16 9 4 1
```

4.5.2.2 for

Syntax

`for start` *`test next body`*

Return

Result of conditionally executed *body*

Description

Like **while**, **for** continues processing until a test expression evaluates to zero. **for** loop execution begins when **start**, which typically initializes variables is executed. When test is evaluated and if true **body** is executed, otherwise the **for** command terminates. After each execution of **body** next which typically modifies **body**, is executed. Test is evaluated again to determine if the for command should continue execution or terminate.

A script you write using **for** can also be written using **while**, but **for** lets you place all of the control code on one line. This aids readability and debugging.

Example

The original script for generating the squares of the first five non-zero integers can be rewritten using **for** as follows:

```
> set y ""
> for {set x 5} {$x > 0} {incr x -1} {
      lappend y [expr $x * $x]
}
25 16 9 4 1
```

4.5.2.3 foreach

Syntax

`foreach` *`var_name list body`*

Return

Result of conditionally executed body

Description

Whereas **while** and **for** let you create loops that act on any variable, including lists, **foreach** is specifically designed to work on lists.

foreach executes body once for each element of the list, e.g., if there are five elements in a list, body is executed five times, once for each element in the list. Before each execution, the current list item is assigned to var-name.

Example

The script for creating a list containing, in decreasing order, the squares of the first five non-zero integers can be rewritten as follows:

```
> set x [list 1 2 3 4 5]
> set y ""
> foreach i $x {
      set y [linsert $y 0 [expr $i * $i]]
}
> set y
25 16 9 4 1
```

With a slight editing, we can change this script so that the generated list will contain the squares of the integers in increasing order.

```
> set x [list 1 2 3 4 5]
> set z ""
> foreach i $x {
      lappend z [expr $i * $i]
}
> set z
1 4 9 16 25
```

4.5.3 Looping Control

You can use the **break** and **continue** commands to abort part or all of a looping command. **break** and **continue** are similar to the corresponding C statements.

4.5.3.1 break

break immediately terminates the innermost enclosing looping command. For example, suppose the list in the examples above contained the first 10 non-zero integers but you want to generate a list that contains only the squares of the first six non-zero integers. You could then rewrite the script as follows, using **break** as the **then** script for an **if** command:

```
> set x [list 1 2 3 4 5 6 7 8 9 10]
> set z ""
> foreach i $x {
      if {$i > 6} break
      lappend z [expr $i * $i]
}
> set z
1 4 9 16 25
```

This second example rewrites the first example from Section 4.5.2.1, terminating the script when the variable *x* equals 3.

```
> set x 5
> set y ""
> while {$x > 0} {
      if {$x == 3} break
      lappend y [expr $x * $x]
      incr x -1
}
> set z
25 16
```

4.5.3.2 continue

continue terminates only the current iteration of the innermost loop. This time, instead of generating the squares of only the first five non-zero integers, we could write a script like the following to omit the square of the sixth integer:

```
> set x [list 1 2 3 4 5 6 7 8 9 10]
> set z ""
> foreach i $x {
      if {$i == 6} continue
      lappend z [expr $i * $i]
}
> set z
1 4 9 16 25 49 64 81 100
```

When using **while**, Tcl skips out of the body and reevaluates the termination expression. When using **for**, next is executed before re-evaluating test. This example rewrites the example from Section 4.5.2.2, skipping the square of the third integer:

```
> set y ""
> for {set x 5} {$x > 0} {incr x -1} {
      if {$x == 3} continue
      lappend y [expr $x * $x]
}
> set y
25 16 4 1
```

4.5.4 switch

Syntax

```
switch ?options? string {pattern body ?pattern body? …}
switch ?options? string pattern body ?pattern body? …
```

Return

Result of conditionally executed body

Description

The **switch** command matches a test string ‘against one or more patterns and executes a corresponding body. Essentially, **switch** is a compact way of writing an **if** command with many **elseif** clauses.

The string argument is usually the contents of a variable, is the value to be matched. Each pattern and body form an argument pair. **switch** iterates through each argument pair until a pattern matches the string, at which time it executes the corresponding body. If **switch** finds no matches, it returns an empty string.

> **NOTE —** You can avoid generating an empty string by using *default* as the last pattern, which matches any value. The corresponding body will be executed if there are no other matches.

The options are **-glob**, **-exact**, and **-regexp**, the same as for **lsearch**. (See Section 4.4.4.) The default is **-glob**.

Example

Both of the following examples do the same thing: it generates the natural logarithm of *$a* if *$x* matches *z1*, the natural logarithm of *$b* if *$x* matches *z2*, the natural logarithm of *$c* if *$x* matches *z3*, or an empty string of there is no match.

```
switch $x {z1 {log $a} z2 {log $b} z3 {log $c}}
switch $x z1 {log $a} z2 {log $b} z3 {log $c}
```

The first form treats each pattern/body pair as an element in a list; the second form treats them as separate arguments. Each form has its advantages

The first form lets you spread the patterns and bodies across multiple lines like this:

```
switch $x {
      z1 {log $a}
      z2 {log $b}
      z3 {log $c}
}
```

The outer braces prevent the newlines from being treated as command separators.

This is especially advantageous when there are many pattern body pairs.

When using the second form you must use backslash-newlines like this:

```
switch $x \
      z1 {log $a} \
      z2 {log $b} \
      z3 {log $c}
```

The major advantage of the second form is that it's easier to perform substitutions on the pattern arguments.

If you enter "-" for a body, **switch** will execute the body of the following pattern. This means several patterns may execute the same body:

```
switch $x {
      z1 -
      z2 {log $a}
      z3 {log $b}
}
```

In this example, **switch** generates the natural logarithm of *$a* when *$x* is *z1* or *z2* and the natural logarithm of *$b* when *$x* is *z3*.

4.5.5 eval

Syntax

`eval` *`argument ?argument? …`*

Return

Result of a script

Description

The **eval** command lets you generate and execute scripts.

eval concatenates the arguments, using spaces to separate them, and executes the result as a Tcl script. A major use is for generating commands, saving them to a variable, and executing the variable as a script.

Example

You might want to use a variable as a counter. From time to time you might want to reset this counter. Each time, you could type the following command:

```
> set counter 0
```

However, you could save this line to a variable, such as

```
> set cmd "set counter 0"
```

and then execute the following when you want to reset the counter:

```
> eval $cmd
```

4.6 Tcl Error/Exception Procedures

The Tcl command that manages exceptions is **catch**, which can trap the error so that the script does not abort. Tcl also contains several global variables (Section 4.7.3), **errorCode** and **errorInfo**, that return information resulting from errors.

> **NOTE —** In addition to **catch**, there are two more Tcl commands, **error** (see Appendix A.11) and **return** (see Appendix A.48), that are related to exceptions. While these commands do not trap exceptions, they can be used when processing exceptions.

4.6.1 catch

Syntax

```
catch command ?varName?
```

Return

0 the script completed normally
1 an error occurred
2 the **return** command was invoked
3 the **break** command was invoked
4 the **continue** command was invoked

Description

The **catch** command evaluates a command or Tcl script, returning an integer identifying the command's completion status. Instead of aborting, **catch** lets the script continue executing.

If you specify the **varName** argument, the entered variable name assumes the command's return value or error message.

Example

```
> catch {expr 2 * (3 + 4} msg
1
> set msg
unmatched parentheses in expression “2 * (3 + 4”
```

4.6.2 Tcl Error Global Variables

4.6.2.1 errorCode

Syntax

```
set errorCode
```

Return

Error information

Description

The Tcl **errorCode** global variable returns error information in the form of a list of one or more elements, the first element identifying a general class of errors and all other elements identifying class-dependent information.

Example

```
> expr 2 * (3 + 4
unmatched parentheses in expression “2 * (3 + 4”
> set errorCode
NONE
```

NOTE — errorCode is filled by only a few commands, primarily those for file access functions and child processes.

4.6.2.2 errorInfo

Syntax

```
set errorInfo
```

Return

Error information

Description

The Tcl **errorInfo** global variable returns diagnostic information on the most recent error. When an error occurs, Tcl generates a basic error message. **errorInfo** returns a more elaborate message than what is returned by **errorCode**.

Example

```
> expr 2 * (3 + 4
unmatched parentheses in expression “2 * (3 + 4”
> set errorInfo
can’t read “unmatched parentheses in expression “2 * (3 + 4”
    while executing
“expr 2 * (3 + 4””: no such variable
```

Since, in this first example, we are entering the Tcl commands on the command-line, **errorInfo** returns just the reason why an error failed. Often, though, you will save a script and load it using the **source** command. For example, let’s assume we’ve saved the above **expr** command to a file called *script1*.

```
> set x [expr 2 * (3 +4 ]
> source script1
unmatched parentheses in expression “2 * (3 + 4”
> set errorInfo
unmatched parentheses in expression “2 * (3 + 4”
  while executing
“expr 2 * (3 + 4”
  (file “script1” line 1)
  invoked from within
“source script1”
```

Not only does **errorInfo** return the reason why the script failed, but it also returns the name of the script where the error occurred and the line number of the command that caused the failure.

Returning the name of the script where the error occurred can be very important since the error might not occur in the script you load directly, but in one that script loads, such as if we had a script, called *script2*, that contained the following

```
> source script1
```

We would then get the following:

```
> source script2
unmatched parentheses in expression “2 * (3 + 4”
> set errorInfo
unmatched parentheses in expression “2 * (3 + 4”
  while executing
“expr 2 * (3 + 4”
  invoked from within
“set x [expr 2 * (3 + 4]...”
  (file “script1” line 1)
  invoked from within
“source script1”
  (file “script2” line 1)
  invoked from within
“source script2”
```

4.7 Procedures

A Tcl procedure is a script that is used to invoke other Tcl commands. Procedures make it easy for you to package commands so that they can be reused easily.

4.7.1 proc

Syntax

`proc` *`proc_name argList body`*

Description

The Tcl **proc** command can be used to define new Tcl command procedures. Once defined, such procedures can be used in the same manner as other built-in commands. **proc** has the syntax

proc_name can be any name you give the command. ***argList*** is a list

Try typing

```
proc sum { a b } { return [expr $a+$b] }
```

This example defined a new command "**sum**" that takes 2 arguments and returns their sum. The **proc** command itself takes 3 arguments:

1. The name of the new command you are implementing
2. A list specifying the arguments to the new command
3. A body that is in turn a list of Tcl commands.

By default, all variables used within a procedure are local. You can now use **sum** as a new command

```
> sum 2 3
5
```

4.7.2 return

Syntax

```
return ?options? ?string?
```

Description

The **return** command immediately returns (or exits) from a procedure without executing the entire script. If specified, **return** returns the entered *string* value. If ***string*** is not specified an empty string is returned. In the **sum** procedure we created above, the procedure evaluates the expression [**expr** *$a+$b*] and returns the correct value.

> **NOTE —** The options for **return** are detailed in Appendix A.48.

4.7.3 Local and Global Variables

The variables in a Tcl procedures are called **local variables** because they can only be accessed within the procedure; they're different from the variables created outside any procedure which are global. When the procedure returns, the local variables are deleted. Global variables last forever unless explicitly deleted.

For example, let's use the **sum** procedure we created above to add two numbers:

```
> set x 6
> set y 8
> sum $x $y
48
```

$x and *$y* are passed to **sum** as the argument list, and the local variables a and b are initialized with these values. Since *$a* and *$b* are local variables, they are deleted after the procedure returns, as shown in the following:

```
> set a
can't read "a": no such variable
> set b
can't read "b": no such variable
```

Each invocation of the procedure creates a new set of local variables. You can have a procedure create local variables, but only those local variables corresponding to the arguments will have a value when the procedure starts executing. For example, let's rewrite the **sum** procedure to create a local variable.

```
> proc sum {a b} {
      set z [expr $a+$b]
      return $z
}
> sum 5 1.2
6.2
set z
can't read "z": no such variable
```

Because *z* is a local variable to the procedure, it can't be accessed outside of the procedure. But what if you want to use a variable from the calling script in the procedure or you want to use a local variable from the procedure in the calling script? To do this, Tcl supports *global variables*.

Global variables are variables named outside any procedure.

You use the **global** command to make already created global variables accessible within a procedure and to create new global variables. Take for example, the following procedure, which is used to raise an entered value to a specified power:

```
> proc pwr {a} {
      global x z
      set z [expr pow($a,$x)]
}
> set x 5
5
> pwr 3
> set z
243
```

The **global** command above make the global variable x accessible within the procedure and created the global variable z if it did not already exist. You can change the value of *$x*, to generate different powers or, as in the following example, to generate the square roots:

```
> set x [expr 1.0/2.0]
> pwr 144
> set z
12.0
```

4.8 String Manipulation

Tcl supports two pattern matching forms for string manipulation. The first follows the shell file name expansion rules, and is similar to globbing. The second uses regular expressions such as those used for **egrep**.

4.8.1 string match

Syntax

```
string match pattern string
```

Return

1 if **pattern** matches **string**
0 otherwise

Description

string match compares the ***pattern*** with the ***string***. If they match, it returns a true response (1). If they don't, it returns a false response (0) This is called the glob-stye of matching. ***pattern*** and ***string*** must be identical, however, the following special sequences may appear in ***pattern***:

*****	Matches any sequence of characters in ***string***, including a null string.
?	Matches any single character in ***string***.
[*chars***]**	Matches any character in the set given by ***chars***. If a sequence of the form ***x–y*** appears in ***chars***, then any character between ***x*** and ***y***, inclusive, will match.
x	Matches the single character ***x***. This provides a way of avoiding the special interpretation of the characters ***?[]** in ***pattern***.

Example

In this example, **string match** extract each element in the list that contains the letter **e**.

```
> set x {Stella Pollock Picasso Wyeth Lischtenstein}
> set y {}
> foreach i $x {
     if [string match *e* $i] {
           lappend y $i
     }
}
Stella Wyeth Lischtenstein
```

4.8.2 regexp

Syntax

```
regexp ?-indices? ?-nocase? ?--? exp string ?matchVar?
           ?subVar subVar …?
```

Return

1 if **exp** matches all or part of **string**
0 otherwise

Description

The **regexp** command matches the regular expression ***exp*** with the ***string***. If they match, it returns a true response (1). If they don't, it returns a false response (0).

See Appendix A.45 for information on regular expressions.

regexp takes the following switches:

–indices	Changes what is stored in the *MatchVar*s. Instead of storing the matching characters from **string**, each variable will contain a list of two decimal strings giving the indices in *string* of the first and last characters in the matching range of characters.
–nocase	Causes upper-case characters in *string* to be treated as lower case during the matching process.
--	Marks the end of switches. The argument following this one will be treated as *exp* even if it starts with a **–.**

4.9 File Input/Output

Tcl provides **stdio**-style file input/output functions. (See the **stdio** manual page.) The most basic of these is the ability to open and close a file.

NOTE — In addition to **open** and **close**, Tcl provides the following commands for performing input/output functions:

- cd
- eof
- file
- flush
- gets
- glob
- puts
- pwd
- read
- seek
- tell

Complete descriptions of these commands can be found in Appendix A.

4.9.1 open

Syntax

`open` *`fileName ?access? ?permissions?`*

Return

fileId to the open file

Description

The **open** command opens the file, ***fileName***, and returns a unique identifier, ***fileId***, which is synonymous to a MYNAH handle. ***fileId*** takes the form **fileN**, where N is a unique iterated number for each successive ***fileId***.

The ***access*** argument indicates how the file is to be accessed. ***access*** can be either a string in the form that would be passed to the **fopen** library procedure or a list of POSIX access flags.

By default, a file is opened in read-only mode.

In the first form, ***access*** can be one of the following values:

r	Open the file in read-only mode; the file must already exist.
r+	Open the file in read/write mode; the file must already exist.
w	Open the file in write-only mode. Truncate it if it exists. If it doesn't exist, create a new file.
w+	Open the file in read/write mode. Truncate it if it exists. If it doesn't exist, create a new file.
a	Open the file in write-only mode. The file must already exist, and the file is positioned so that new data is appended to the file.
a+	Open the file in read/write mode. If the file doesn't exist, create a new empty file. Set the initial access position to the end of the file.

In the second form, ***access*** consists of a list of any of the following flags.

> **NOTE —** One of the flags must be either **RDONLY**, **WRONLY** or **RDWR**.

RDONLY	Open the file in read-only mode.
WRONLY	Open the file in write-only mode.
RDWR	Open the file in read/write mode.
APPEND	Set the file pointer to the end of the file prior to each write.
CREAT	Create the file if it doesn't already exist (without this flag it is an error for the file not to exist).

EXCL	If **CREAT** is specified also, an error is returned if the file already exists.
NOCTTY	If the file is a terminal device, this flag prevents the file from becoming the controlling terminal of the process.
NONBLOCK	Prevents the process from blocking while opening the file.
TRUNC	If the file exists it is truncated to zero length.

The ***permissions*** argument is used to set the permissions of a file that is created when it is opened, such as when using the ***w+ access*** argument. ***permissions*** defaults to 0644, giving only you read/write permissions for the file but allowing all other users to read the file.

Since a ***fileId*** works the same as a MYNAH handle, you can use the **set** command to assign the ***fileId*** to a variable.

Example

These examples illustrate opening a file using the first ***access*** form. In the first example, you want to open an existing file with the ***fileName*** *Test1* in read/write mode. In the second example, you create and open a file, also in read/write mode, and assign it to a variable.

```
> open Test1 r+
file1

> set x [open Test2 w+]
file2
```

This example illustrates creating and opening a file using the second ***access*** method. In addition, the ***permissions*** argument is used so that only you can read or write the file.

```
> set y [open Test3 {RDWR CREAT} 0600]
file3
```

4.9.2 close

Syntax

`close` *`fileId`*

Return

An empty string

Description

The **close** command simply closes a file previously opened using the **open** command.The input **must** be the ***fileId*** generated by an invocation of the **open** command or a handle assigned to a ***fileId***. You **can not** use **close** on a ***fileName***; the ***fileName*** exists only in the operating system and not in Tcl.

Example

These examples illustrate closing the first two examples of using the **open** command. The first example works directly on the ***fileId*** and the second works on the variable assigned to a fileId.

```
> close file1
> close $x
```

4.10 Using xmytclsh

The **xmytclsh** tool lets you interactively run scripts. **xmytclsh** reads Tcl, TclX, and MYNAH Tcl extensions commands from the standard input or from a file using the **source** command. (See Section 4.11 for information on the **source** command.)

> **NOTE — xmytclsh** is a re-implementation of the Tcl utility **tclsh**, adding support for the TclX and MYNAH extensions.

You may wish to use **xmytclsh** to verify the examples included in this section or to test your own examples.

> **NOTE — xmytclsh** is primarily useful for debugging commands as you incorporate them into scripts. It is ***not*** recommended for running scripts, such as for end-to-end testing, since it does not generate output files.

The syntax is

```
xmytclsh ?-h? ?-d filename? ?-l level?
```

where

-h	Prints this help message
-d *filename*	Turns on tracing. *filename* specifies a trace file (tracing is off unless **-d** is specified). If tracing is on, child script executions will be traced through all processes.
-l *level*	Specifies the level of trace output (off, low, high).

When you type **xmytclsh**, your UNIX system prompt is replaced with a > prompt. You can begin entering your Tcl commands and extensions or import a script.

To exit **xmytclsh** you must use the **exit** or **xmyExit** extensions.

Entering **`Control-C`** in **xmytclsh** interrupts the currently running command (e.g., **xmySleep 20** or a long-running **while** loop) and gives you a clean > prompt.

See Section 2.3 for information on executing script files.

> **NOTE —** The standard Tcl package supports three predefined variables that aid in processing command-line arguments, such as when invoking a Tcl script: **argv0**, **argv**, and **argc**. **argv0** stores the name of the script. **argv** stores the command-line arguments. **argc** stores the number of command line arguments. These variables are not defined for **xmytclsh**.

4.11 Importing Scripts Using the source Command

As you accumulate scripts, you may find that you are creating lines of script, such as a procedure that can be used in serveral scripts. Rather than entering these lines in each script, you can save them into files, and then import the files into the new script using the **source** command.

When using **xmytclsh**, you can also use **source** to load and execute an entire script, however, since this does not generate any of the MYNAH reports it is not very useful.

Syntax

`source` *`script`*

Description

The **source** command lets you read in and execute a Tcl script you've saved to a file. For example, you could use an editor to create a file, e.g. *test1*, containing the following lines:

```
> set x {4 9 25}
> set y [sqrt[join $x *]]
```

You could then type

```
> source test1
30
```

5. xmyVar Global Script Variables

This section contains the complete list of the variables in the global **xmyVar** array. The variables are read-write except where indicated. These are array elements that are available to all domains.

One of the variables is the MYNAH symbol table, which is described in more detail in Sections 5.17, 6.2.17, 6.2.18, 6.2.19, and 6.2.20.

5.1 Channel

Syntax

```
set xmyVar(Channel)
```

Description

xmyVar(Channel) returns the Telexel channel name of the process. This is the communications channel on which the Script Engine receives all its messages.

This variable is read-only.

5.2 DatabaseMode

Syntax

```
set xmyVar(DatabaseMode)
```

Description

xmyVar(DatabaseMode) is set to true (1) if the SE is running in database mode, false (0) otherwise. For MYNAH to be running in the database mode means the database is up and MYNAH is interacting with it when it runs scripts.

This variable is read-only.

Example

A basic use is to determine whether you are in database mode or not.

```
> set xmyVar(DatabaseMode)
1
```

In this example, you test to see if you are in database mode. If you are not, you exit. If you are, you create a connection.

```
> if {$xmyVar(DatabaseMode) = 0} {
   xmyExit "Exiting, not in database mode"
} else {
   set Conn [xmyTerm3270 connect]
}
```

5.3 EngineMode

Syntax

```
set xmyVar(EngineMode)
```

Description

xmyVar(EngineMode) returns one of **FullState**, **ConnOnly**, or **StateLess**. Command-line SEs always run in **FullState** mode. Background and embedded SEs run in any of the three modes.

This variable is read-only.

Example

The simplest example is to return the engine mode.

```
> set xmyVar(EngineMode)
fullState
```

In this example, you reconnect only if you are in stateless mode to save overhead involved in reconnecting.

```
> if {$xmyVar(EngineMode) == "stateLess"} {
   set Conn [xmyTerm3270 connect]
   xmySymTblPut -tag Conn -value $Conn
} else {
   set Conn [xmySymTblGet -tag Conn]
}
```

5.4 EngineType

Syntax

```
set xmyVar(EngineType)
```

Description

xmyVar(EngineType) is set to one of the three SE types: **embedded**, **background**, or **commandline**.

This variable is read-only.

Example

You can use **xmyVar(EngineType)** to find out what SE type is being used.

```
> set xmyVar(EngineType)
commandline
```

This example prompts you for a password if you are running the script interactively. If not, the script uses the default password.

```
> if {$xmyVar(EngineType) == "embedded"} {
   keylset v -prompt "Enter Password"
   set pword [xmyPrompt [list $v]]
} else {
   set pword "default"
}
```

5.5 ExitHandler

Syntax

```
set xmyVar(ExitHandler) Handler_procedure
```

Description

xmyVar(ExitHandler) takes as input the name of a Tcl procedure to call immediately after the script normally or abnormally terminates. The Tcl procedure should have the following syntax:

```
proc myExitHandler {code returnValue} {}
```

where

code	Specifies the return code of the script. Usually this is the string ***ok*** or ***error***, but can also be any acceptable ***code*** of the **return -code *code*** command (see Appendix A.48).
returnValue	Specifies the last interpreter result value.

As long as the Tcl interpreter is not corrupted during script execution, this function is always called.

> **NOTE —** The **ExitHandler** function is called only when executing a script in the background. It is not called if the script is executed using **xmytclsh** or the GUI's embedded SE.

An **ExitHandler** can influence the Run Summary field in the Runtime object. This is important because the Run Summary field is one of the columns in the Job Status window of the GUI and is therefore always visible to users.

It is recommended that users write an **ExitHandler** procedure. Once the procedure is written, it can be made use of in one of three ways:

- Each script can have a **set xmyVar(ExitHandler)** line
- Each script can call a user written initialization procedure and the initialization procedure has the **set xmyVar(ExitHandler)** line
- Configure SE groups to run a start-up script and the start-up script has the **set xmyVar(ExitHandler)** line.

Since the **ExitHandler** is evaluated immediately after the script finishes, it masks the final run status of the script and the interpreter result unless the procedure propagates them upward to the script submitter using the **return** command, as in the first example below.

When the **xmyExit** method is used in the **ExitHandler**, the script's final return code will always be "OK." If the prior return code (***$code***) is needed, the argument to **xmyExit** should include the code, as in the third example below.

Examples

The following example uses the return command to propagate the script's return code and the interpreter's result back to the caller:

```
> proc myExitHandler {code returnValue} {
   if {$code != "ok"} {
      exec echo "$xmyVar(ScriptName) failed: $returnValue" \
               | mailx user@sys
   }
   return -code $code $returnValue
}

> set xmyVar(ExitHandler) myexitHandler
```

The following example determines what string should be returned. (The Run Summary field of the **Runtime** object will be set to this string.) Note that neither parameter to the procedure is returned.

```
> proc exitHandler {code returnValue} {
   global xmyVar
   if {$xmyVar(FailedCompares) > 0 ||
      $xmyVar(WarningCompares) > 0} {
      xmyExit "inconclusive"
   } else {
      xmyExit "success"
   }

}> set xmyVar(ExitHandler) myexitHandler
```

This last example illustrates how to include the return code (***$code***) and the interpreter's results (***$returnValue***) in the **xmyExit** string to be returned.

```
> proc exitHandler {code returnValue} {
   global xmyVar
   xmyExit "code=$code : returnValue=$returnValue"
}

> set xmyVar(ExitHandler) xmyexitHandler
```

5.6 FailedCompares

Syntax

```
set xmyVar(FailedCompares)
```

Description

xmyVar(FailedCompares) is the number of compares that have failed in the currently executing script.

This variable is read-only.

There are several MYNAH language commands that automatically update **xmyVar(FailedCompares)**. The following list identifies the commands according to their extension packages:

- General package
  - **— xmyCompare**
  - **— xmyDiff**
- TermAsync package
  - **— $connection compare**
- Term3270 Package
  - **— $connection compare**
- TermFCIF package
  - **— $handle compare**
  - **— $handle compareTags**
  - **— $handle extraTags**

Each of these commands will also produce **compare** blocks in the *compares* file.

Example

In this example, you expect to find four failed comparisons. If this is not true, you have the script return a string stating how many failed comparisons were found.

```
> if { $xmyVar(FailedCompares) != 4 } {
   set result "$result|failed compares should be 4,
         instead $xmyVar(FailedCompares)"
}
```

This time you check to see if there were any failed comparisons. If there were, you exit with the exit string *failure*. If there weren't, you exit with the exit string *success*.

```
> if {$xmyVar(FailedCompares) > 0} {
   xmyExit "failure"
} else {
   xmyExit "success"
}
```

5.7 GoodCompares

Syntax

```
set xmyVar(GoodCompares)
```

Description

xmyVar(GoodCompares) is the number of compares that have been executed successfully in the currently executing script.

This variable is read-only.

There are several MYNAH language commands that automatically update **xmyVar(GoodCompares)**. The following list identifies the commands according to their extension packages:

- General package
  - **— xmyCompare**
  - **— xmyDiff**
- TermAsync package
  - **— $connection compare**
- Term3270 Package
  - **— $connection compare**
- TermFCIF package
  - **— $handle compare**
  - **— $handle compareTags**
  - **— $handle extraTags**

Each of these commands will also produce **compare** blocks in the *compares* file.

Example

In this example, you expect to find six successful comparisons. If this is not true, you have the script return a string stating how many successful comparisons were found.

```
> if { $xmyVar(GoodCompares) != 6 } {
   set result "$result|good compares should be 6, \
         instead $xmyVar(GoodCompares)"
}
```

5.8 LibraryPath

Syntax

```
set xmyVar(LibraryPath)
```

Description

xmyVar(LibraryPath) returns the value of the SE configuration **LibraryPath** parameter.

This variable is read-only.

5.9 MaxFails

Syntax

```
set xmyVar(MaxFails)
```

Description

xmyVar(MaxFails) is used to set the maximum number of **compare** statements allowed to fail before the script exits. The script will abort when the **xmyVar(MaxFails)** value is reached.

Example

The following sets the maximum number of **compare** statements so that a script will abort when when two failed **compare** statements have occured.

```
> set xmyVar(MaxFails) 2
```

5.10 MaxFailsHandler

Syntax

```
set xmyVar(MaxFailsHandler)
```

Description

xmyVar(MaxFailsHandler) takes as input the name of a Tcl procedure to call when the number of failed compares equals **xmyVar(MaxFails)**. The Tcl procedure should have the following syntax:

```
> proc myFailsHandler {} {
   MaxFailsHandler script
}
```

This function executes in the same context of the script, so system and user variables are all available. For example, the following is a valid handler:

```
> set xmyVar(MaxFailsHandler) myFailsHandler
> proc myFailsHandler {} {
      global xmyVar
      exit $xmyVar(FailedCompares)
}
```

xmyVar is declared as global so that when the script terminates, ***xmyVar*** will pass the number of failed comparisons.

> **NOTE —** Remember, the script will abort when the number of failed compares set by **xmyVar(MaxFails)** is reached.

Example

In the following, we first define the procedure **maxhandler**. The return string from **maxhandler**'s script, the string *max fails exceeded*, is returned to the caller of the script.

```
> proc maxhandler {} {
   xmyExit "max fails exceeded"
}
```

```
> set xmyVar(MaxFails) 2
> set xmyVar(MaxFailsHandler) maxhandler
> set xmyVar(OutputLevel) {*}
```

5.11 OutputDir

Syntax

```
set xmyVar(OutputDir)
```

Description

xmyVar(OutputDir) is the name of the directory that is to contain script output.

This variable is read-only.

Example

In this example you create your own file, *userData*, in the output directory.

```
> set fileName $xmyVar(OutputDir)/userData
> set filePtr [open $fileName w]
> foreach item $alist {
   puts $filePtr $item
}
> close $filePtr
```

5.12 OutputLevel

Syntax

```
set xmyVar(OutputLevel)
```

Description

xmyVar(OutputLevel) is a Tcl list containing the type of output that should be written to the script output directory, e.g., { *.* }.

Example

In this example you use the star (*) wildcard to write all output to the output directory.

```
> set xmyVar(OutputLevel) {*}
```

5.13 RuntimeId

Syntax

```
set xmyVar(RuntimeId)
```

Description

xmyVar(RuntimeId) is the ID of the database Runtime Object used to record the status of the current script.

This variable is read-only.

```
> set xmyVar(RuntimeId)
43
```

5.14 ScriptName

Syntax

```
set xmyVar(ScriptName)
```

Description

xmyVar(ScriptName) is the full name of the script being executed, e.g., */home/scripts/script1.tcl*.

This variable is read-only.

Example

```
> set xmyVar(ScriptName)
/home/scripts/script1.tcl
```

5.15 SEGroup

Syntax

```
set xmyVar(SEGroup)
```

Description

xmyVar(SEGroup) is the name of the SE group to which the SE running the script belongs. Embedded SEs and Command-line SEs have a group name of "".

This variable is read-only.

Example

```
> set xmyVar(SEGroup)
SeGp1
```

5.16 SubmittedBy

Syntax

```
set xmyVar(SubmittedBy)
```

Description

xmyVar(SubmittedBy) contains a UNIX username. In Command-line and Embedded SEs, **xmyVar(SubmittedBy)** contains the UNIX username of the person who started the process. In Background SEs, **xmyVar(SubmittedBy)** contains the UNIX username of the person who submitted the script (via the GUI or CLUI).

This variable is read-only.

Example

In this example, a different password is used depending on who ran the script. The file ‘logins’ contains test logins and passwords.

```
> set l [xmyUdb read -file $env(XMYHOME)/logins -decrypt]
> set aPassword [keylget l $xmyVar(SubmittedBy)]
> if {$aPassword == ""} {
   xmyExit "no password found for user $xmyVar(SubmittedBy)"
} else {
   LoginToSut $xmyVar(SubmittedBy) $aPassword
}
```

5.17 SymTbl

Syntax

```
set xmyVar(SymTbl)
```

Description

xmyVar(SymTbl) is the MYNAH symbol table passed into scripts at start-up and passed back to the execution requester at script termination time.

This variable is read-only.

The format of the symbol table is a Tcl list of lists, where each sublist contains a variable (tag)/value pair. Like any Tcl list, its size is limited only by the amount of memory available to the process in which it is executed.

During the execution of the parent script, the symbol table is passed along with requests for child script execution. New or changed values are returned when the results of child execution are handled (after **sendWait** or **Wait** commands). New or changed values are also passed to the original sender of the parent script execution request at completion time.

Entries in the symbol table can be created using **xmySymTblPut** (Section 6.2.20). **xmyVar(SymTbl)** will return the entire symbol table while **xmySymTblGet** (Section 6.2.19) will return specific values for a spcecified tag.

Example

```
> set Conn [xmyTerm3270 connect]
.xmyTerm3270_1

> xmySymTblPut -tag Conn -value $Conn
> set xmyVar(SymTbl)
{Conn .xmyTerm3270_1}

> xmySymTblPut -tag test -value 42
> set xmyVar(SymTbl)
{Conn .xmyTerm3270_1} {test 42}
```

5.18 SymTblNAC

Syntax

```
set xmyVar(SymTblNAC)
```

Description

xmyVar(SymTblNAC) is an internal list not useful to users.

This variable is read-only.

5.19 TestVersionId

Syntax

```
set xmyVar(TestVersionId)
```

Description

xmyVar(TestVersionId) is the current unique Test Version ID based on the scoping of **xmyBegin/xmyEnd** statements.

This variable is read-only.

5.20 TimeoutHandler

Syntax

```
set xmyVar(TimeoutHandler)
```

Description

xmyVar(TimeoutHandler) is the name of a Tcl procedure to call when there is no response from the SUT in the number of seconds specified at the domain or connection level. The Tcl procedure should have the following syntax:

```
proc TimeoutHandler {conn pkg} {}
```

where

conn is the handle of the connection on which the timeout occurred.

pkg is the domain extension package identifier (e.g., **xmyTermAsync**).

If no **TimeoutHandler** is defined and a timeout occurs, the script exits with the status TCL_ERROR. If a **TimeoutHandler** is defined, the engine will execute the timeout handler before replying to the requester of script execution. The reply status in this case will be TCL_OK if the **TimeoutHandler** executed successfully. To override this behavior and return TCL_ERROR regardless of whether a **TimeoutHandler** is defined, a handler similar to the one in the example can be used.

NOTE — Timeouts are set at the domain level.

Example

This example returns an error so that the requester of script execution sees a status of TCL_ERROR.

```
> proc myTimeoutHandler {conn pkg} {
   return -code error "timeout in conn $conn, pkg $pkg"
}
```

5.21 UpdateCompares

Syntax

```
set xmyVar(UpdateCompares)
```

Description

xmyVar(UpdateCompares) is the flag that tells the MYNAH System whether to update counters. When this variable is set to **true** (1), which is the default, **compare** statements update the value of the variables **xmyVar(GoodCompares)**, **xmyVar(FailedCompares)**, and **xmyVar(WarningCompares)**.

Example

In this example, **xmyVar(UpdateCompares)** is set to false, letting you run some compares with without updating the **GoodCompares**, **FailedCompares**, and **WarningCompares** variables.

```
> set xmyVar(UpdateCompares) false

> xmyCompare -expr {$x==5}
> xmyCompare -expr {$x==6}

> xmyCompare -expr {$x==5} -label label5
> xmyCompare -expr {$x==6} -label label6

> xmyCompare -expr {$x==5} -warning
> xmyCompare -expr {$x==6} -warning
```

5.22 WarningCompares

Syntax

```
set xmyVar(WarningCompares)
```

Description

xmyVar(WarningCompares) is the number of warning (not failed) compares executed in the current script. This count is incremented when the **-warning** option is used in **compare** methods.

This variable is read-only.

There are several MYNAH language commands that automatically update **xmyVar(WarningCompares)**. The following list identifies the commands according to their extension packages:

- General package
  - **xmyCompare**
  - **xmyDiff**
- TermAsync package
  - **$connection compare**
- Term3270 Package
  - **$connection compare**
- TermFCIF package
  - **$handle compare**
  - **$handle compareTags**
  - **$handle extraTags**

Each of these commands will also produce **compares** blocks in the *compares* file.

Example

In this example, you expect to find two warning comparisons. If this is not true, you have the script return a string stating how many warning comparisons were found.

```
> if { $xmyVar(WarningCompares) != 6 } {
   set result "$result|warning compares should be 2,
         instead $xmyVar(WarningCompares)"
}
```

6. General MYNAH Tcl Extensions

6.1 Overview

The Tcl interpreter in a Script Engine (SE) contains the general MYNAH extension package, which implements a set of Tcl commands that are automatically available to a script when it starts executing.

6.2 General Commands

Table 6-1 lists the general MYNAH extensions, grouping them in the categories listed in Table 1-3.

Table 6-1. General MYNAH Extensions (Sheet 1 of 2)

Category	Command	Description	Section
Connection	**exit**	Re-implementation of the Tcl **exit** command to act like **xmyExit**.	6.2.1, Page 6–3
	xmyExit	Explicitly exits the script and the Tcl interpreter.	6.2.7, Page 6–15
	xmyLoadPkg	Loads a Tcl extension package.	6.2.9, Page 6–17
	xmyUnloadPkg	Removes the named extension package from the SE.	6.2.21, Page 6–36
Data Entry/Retrieval	**xmyDate**	Returns the requested date and time.	6.2.4, Page 6–7
	xmyPrint	Writes to the output file in the script output directory.	6.2.11, Page 6–23
	xmyPrompt	Allows an embedded script to receive data from you at script execution.	6.2.12, Page 6–24
	xmySymTblExists	Determines if a variable exists in the MYNAH symbol table.	6.2.18, Page 6–33
	xmySymTblGet	Retrieves the value of a variable from the symbol table.	6.2.19, Page 6–34
	xmySymTblPut	Lets you add or change values in the symbol table.	6.2.20, Page 6–35
	xmySymTblDel	Remove the tag/value pair reference from symbol table.	6.2.17, Page 6–32

Table 6-1. General MYNAH Extensions (Sheet 2 of 2)

Category	Command	Description	Section
Comparisons	**xmyBegin**	Begins a block that delimits a test.	6.2.2, Page 6–4
	xmyCompare	Updates the script variables **xmyVar(GoodCompares)** or **xmyVar(FailedCompares)** or **xmyVar(Warningcompares).**	6.2.3, Page 6–6
	xmyDiff	Compares two files.	6.2.5, Page 6–10
	xmyEnd	Terminates a block that delimits a test.	6.2.6, Page 6–14
	xmyMask	Class command used to produce an instance of a mask object.	6.2.10, Page 6–19
	xmyReadGrep	Reads the next line from a file that matches a regular expression.	6.2.13, Page 6–25
	xmyRegex	Searches for a regular expression in the given input data string.	6.2.14, Page 6–28
	xmySimilar	Compares numbers to see if they differ by an entered percentage.	6.2.15, Page 6–30
	xmyUpdateResult	Updates the status for the Result object	6.2.22, Page 6–37
Location	**xmyGetLine**	Returns a line corresponding to an entered integer.	6.2.8, Page 6–16
Waiting	**xmySleep**	Pauses the script	6.2.16, Page 6–31

6.2.1 exit

Syntax

```
exit ?exitString?
```

Return

No result

Description

The standard Tcl **exit** command is re-implemented to behave exactly like **xmyExit**. See Section 6.2.7, for more information.

Example

```
> exit “too many failed compares”
```

6.2.2 xmyBegin

Syntax

```
xmyBegin -testid id
xmyBegin -label label
```

Return

No result

Description

The **xmyBegin** command starts a block that delimits a test. **xmyBegin** is used when database results reporting is required. (i.e., Test Management.) An **xmyBegin/xmyEnd** block collects compare results for that block and associates them with a test.

If MYNAH is running in non-database mode, **xmyBegin** does nothing (including syntax checking.

There are two version of this command:

- The ***id*** version starts a block that delimits a test. The ***id*** is inserted in the code by the user, who gets the ***id*** from the database. The ID was generated when the user created the test object. It represents a test version object. **xmyBegin** must be terminated *in the same script* by **xmyEnd**. If not, incomplete results will be reported to the database.

  NOTE — See Section 8 of the *MYNAH System Users Guide*, for information on Test Objects.

- The ***label*** version must be enclosed in an **xmyBegin/xmyEnd** block with a testid. It provides a runtime-defined grouping of compares.

  Labels need not be unique and can be nested.

It is an error for two **xmyBegin** commands to have the same ***testid***. The SE maintains a list of all ***testids*** encountered to enforce this restriction.

Example

```
> xmyBegin -testid 2334
> if {$xmyVar(FailedCompares) == 0} {
    xmyExit "my result=ok"
} else {
    xmyExit "my result=fail"
}
> xmyEnd -testid 2334
```

Exceptions

If **xmyBegin -label** ***label*** is not preceded by **xmyBegin -id** ***id*** at some point in the script.

Two **xmyBegin** commands with the same ***testid***.

6.2.3 xmyCompare

Syntax

```
xmyCompare -expr arbitraryExpression ?-label label? \
           ?-warning?
```

Return

The result of evaluating ***arbitraryExpression*** (1 or 0).

Description

The **xmyCompare** command updates the script variables **xmyVar(GoodCompares)** or **xmyVar(FailedCompares)**, depending on the result of evaluating ***arbitraryExpression***. If the **xmyCompare** method is contained inside a begin/end block (see **xmyBegin**), the results for that block are also updated.

xmyCompare takes the following options:

-expr ***arbitraryExpression***	Indicates the expression to be evaluated.
-label *label*	Indicates that this is a labeled compare. The label will be written to the script output file and saved in the **CompareResult** database object.
-warning	Indicates that the result of compare should update **WarningCompares** instead of the **FailedCompares**.

Side Effects

Writes compare event to script output and updates script variables listed in the description.

Example

Script command (where **$result** would have been set somewhere in the code):

```
> xmyCompare -expr {$result==”ok”}
```

Script output event (if compare succeeded):

```
19951201:125637:compare:General:data::good:<some-index-number>
```

The index numbers contain the number of bytes into the *compares* file to look for the record of that **compare**. The numbers are indexes into the *compares* file in the output directory.

6.2.4 xmyDate

Syntax

```
xmyDate ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -month ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -monthName ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -day ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -dayName ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -year ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -hour ?-time timestring? ?-increment n? ?-decrement n?
xmyDate -minute ?-time timestring? ?-increment n? \
                                    ?-decrement n?
xmyDate -second ?-time timestring? ?-increment n? \
                                    ?-decrement n?
xmyDate -julian ?-time timestring? ?-increment n? \
                                    ?-decrement n?
```

Return

Date or time corresponding to entered arguments.

Description

The **xmyDate** command returns the requested date and time functionality as determined by the provided arguments.

> **NOTE —** The descriptions and examples in this section assume today is Wednesday, August 23, 1995, 10:00 AM EDT.

xmyDate returns a date or time as determined by the following arguments:

1. If no arguments are provided, **xmyDate** returns the current date in seconds since 00:00:00 GMT, January 1, 1970.

   If **-time** is provided, **xmyDate** converts the **timestring** following **-time** to seconds since 00:00:00 GMT, January 1, 1970 instead of using the current date.

   If either **-increment** or **-decrement** option is provided followed by a value, the current date in seconds since 00:00:00 GMT, January 1, 1970 increased or decrease by the appropriate value is returned.

2. If **-month** is provided, **xmyDate** returns the current month (1 - 12).

NOTE — The TclX **fmtclock** command returns a three character string for the month. For example, **getclock** returns
809186400.
As input to **fmtclock**, it returns
Wed Aug 23 10:00:00 EDT 1995.

3. If **-monthName** is provided, **xmyDate** returns the full name of the current month.
4. If **-day** is provided, **xmyDate** returns the current day of the month (1 - 31).
5. If **-dayName** is provided, **xmyDate** returns the current day as a three character string.
6. If **-year** is provided, **xmyDate** returns the current year as a four character string.
7. If **-hour** is provided, **xmyDate** returns the current hour as a two character string.
8. If **-minute** is provided, **xmyDate** returns the current minute as a two character string.
9. If **-second** is provided, **xmyDate** returns the current second as a two character string.
10. If **-julian** is provided, **xmyDate** returns the Julian date equivalent of the current date.

For Items 2 through 8, **xmyDate** first determines whether an optional **-time** argument followed by **timestring** is present. If it is, **timestring** is used instead of the current date and time. The appropriate field is then extracted or the Julian calculation performed. For example, if the argument **month** is used in the **xmyDate** command then the **month** part of the **timestring** is extracted and used for the **xmyDate** command.

You can also use either the **-increment** or **-decrement** option followed by a value. If one is present, then the appropriate arithmetic is performed on the extracted field, and the calculated value is returned. If neither is present, the extracted field is returned.

For Item 10, either the current date or the value following **-time** is converted to a Julian date and then the **-increment** or **-decrement** arithmetic is performed and the result returned.

If **-year** is provided, **xmyDate** returns the current year as a four character string.

> **NOTE —** If a timestamp contains a two character year, **xmyDate** determines the century, based on wether the year is in the range 70-99 or the range 00-38. Any year in the range 70-99 is assumed to mean 1970-1999, and any year in the range of 00-38 is assumed to mean 2000-2038.

> **NOTE —** Values in the range 39-69 are NOT supported.

For more information on permissible formats for “timestring” see Appendix B.5.5.

Example

This example returns the seconds since 00:00:00 GMT, January 1, 1970.

```
> xmyDate
809186400
```

This example calculates the current month, August (08) and increments it by five, January (01).

```
> xmyDate -month -increment 5
01
```

This example increments the current date in seconds since 00:00:00 GMT, January 1, 1970 by five seconds.

```
> xmyDate -increment 5
809186405
```

This example increments the number of seconds between 00:00:00 GMT, January 1, 1970 and the entered timestring, the last second of 1995, by five seconds.

```
> xmyDate -time {Dec 31, 1995 11:59:59 PM EST} -increment 5
820472404
```

This example increments the month of December by one, returning **01** (for January)

```
> xmyDate -month -time {Dec 1} -increment 1
01
```

This example increments the month of December by one, returning the name January.

```
> xmyDate -monthName -time {Dec 1} -increment 1
January.
```

This example returns the Julian date equivalent of the current date.

```
> xmyDate -julian
235
```

For more information on permissible formats for ***timestring*** see Appendix B.5.5.

6.2.5 xmyDiff

Syntax

```
xmyDiff ?-label label? ?-warning? ?-noFormat? \
           ?-listFile listFile? \
           ?-sedScriptFile sedScriptFile? \
           ?-scriptBaseFile scriptBaseFile?

xmyDiff ?-label label? ?-warning? ?-noFormat? \
           ?-listFile listFile? \
           ?-sedScriptFile sedScriptFile? \
           -file1 file1 -file2 file2
```

Return

1 if there were differences

0 if there were no differences

Description

xmyDiff's first form compares ***scriptBaseFile.out*** with ***scriptBaseFile.mstr***. **xmyDiff** assumes *scriptBaseFile.out* is in the current script output directory and that *scriptBaseFile.mstr* is in the same directory as the current script.

xmyDiff's second form compares ***file1*** with ***file2***.

Both forms take the following options:

-label *label*	Indicates that this is a labeled compare. The label will be written to the script output file and saved in the **CompareResult** database object.
-warning	Causes the script variable **xmyVar(WarningCompares)** to be incremented instead of **xmyVar(FailedCompares)** if differences are found.
-noFormat	If this option is specified, the files to be compared will not be formatted first. This option speeds up processing of non-FCIF messages.

-listFile ***listFile***	Specifies the name of a file containing a list of **sed(1)** scripts to be applied to the files to be differenced. If ***listFile*** does not start with '/', MYNAH will assume it resides in the $XMYHOME/data/sedscripts directory. Furthermore, if any of the sed script files it contains do not start with '/', MYNAH will assume those files reside in the $XMYHOME/data/sedscripts directory. ***listFile*** can be a list of file names instead of a single file name. This option can be used in combination with the **-sedScriptFile** option.
-sedScriptFile ***sedScriptFile***	Specifies the name of the **sed(1)** script to be applied to the files to be differenced. If ***sedScriptFile*** does not start with '/', MYNAH will assume it resides in the $XMYHOME/data/sedscripts directory. This option can be used in combination with the **-listFile** option. ***sedScriptFile*** can be a list of file names instead of a single file name.

xmyDiff's first form also takes the following option:

-scriptBaseFile ***scriptBaseFile***	Specifies the name of ***scriptBaseFile.out***. If *scriptBaseFile* is not specified, the default value is the name of the script stripped of its *.tcl* suffix.

xmyDiff's second form also takes the following options:

-file1 ***file1***	Specifies the name of the first file in the comparison.
-file2 ***file2***	Specifies the name of the file to compared with ***file1***.

Files generated by the Term3270 package may contain the UNIX carriage return character, ^M (<control>-M), at the end of each line. The MYNAH System includes an **sed** script, ***strip_ctl_m***, to strip out the ^M characters from these files when **xmyDiff** compares them. This script is located in *$XMYHOME/data/sedscripts* and can be specified as the ***sedScriptFile*** to the **-sedScriptFile** option.

For example, if you are comparing *file1* with *file2* and either *file1* or *file2* has ^M characters that should be stripped out prior to the compare operation, you would execute

```
xmyDiff -sedScriptFile strip_ctl_m file1 file2
```

> **NOTE —** If your files have control-M characters in them, you are strongly advised to strip them out using

-sedScriptFile *strip_ctl_m* as shown above, otherwise you will get unpredictable results.

NOTE — Since the **strip_ctl_m sed** script resides in the *$XMYHOME/data/sedscripts* directory, the **xmyDiff** subcommand will find it automatically and you don't have to specify the full directory.

Exceptions

Missing/invalid arguments

Failure to open temporary work file in */tmp*

Failure to open any specified **sed** script file

Failure to open any specified list file

Combining **-file1** and **-file2** arguments with **-scriptBaseFile** argument

Failure to open **-file1** or **-file2**

Specifying only one of **-file1** or **-file2**

Failure to determine the script base file name

Failure to determine the master file name

Failure to open the master file

Failure to determine the output file name

Failure to open the output file

Failure to fork **xmyDiff** shell process

Exit failure in **xmyDiff** shell process

Side Effects

Updates the values of the script variables **xmyVar(GoodCompares)**, **xmyVar(FailedCompares)**, and **xmyVar(WarningCompares)**.

Creates an event in the script output file and adds differenced records to the *compares* file.

> **NOTE —** When an exception occurs, the compare events are not recorded in the script output file or the script compare file.

Example

Script command:

```
> xmyDiff -warning -scriptBaseFile script1
```

Script output event (if no differences were found):

```
19960523:141014:compare:General:diff::good - no differences:2024
```

6.2.6 xmyEnd

Syntax

```
xmyEnd -testid id
xmyEnd -label label
```

Return

No result

Description

The **xmyEnd** command terminates a block that delimits a test. The argument must correspond to an ***id*** or ***label*** argument to an earlier **xmyBegin** command. If it corresponds to an ***id***, then the results object for that test object will be updated at script completion time. If the begin/end block is nested inside another begin/end block, the compare counts (the values of **goodCompares**, **failedCompares**, and **warningCompares**) are added to the compare counts of the outside block.

When **xmyEnd** is executed, the results for that block are saved and flagged as a database update that must be performed at script completion time.

Exceptions

If no corresponding **xmyBegin**.

6.2.7 xmyExit

Syntax

`xmyExit` *`?exitString?`*

Return

No result

Description

The **xmyExit** command explicitly exits the script. It also exits the Tcl interpreter unless the SE is running in **fullState** mode. **xmyExit** returns the exit code **TCL_OK** and the value of ***exitString***. If ***exitString*** is not specified, the previous value is used. The Tcl **error** command must be used to return a code other than **TCL_OK**.

Command line SEs write the ***exitString*** to `stdout`. Embedded SEs return it to the invoker.

> **NOTE —** If the **xmyVar(ExitHandler)** variable (Section 5.5) is set, the ***exitString*** value is ignored when the script ends.

Side Effects

Write summary events to script output.

6.2.8 xmyGetLine

Syntax

```
xmyGetLine -text string -lineNumber integer
```

Return

A string containing the requested data or the null string if ***string*** is null or ***integer*** is out of range.

Description

The **xmyGetLine** command returns the “line” of ***string*** corresponding to ***integer***. **xmyGetLine** expects ***string*** to contain embedded newline characters. Line numbers start at 1.

Example

```
> set thirdLine [xmyGetLine -text “first\nsecond\nthird\n”
      -lineNumber 3]
```

6.2.9 xmyLoadPkg

Syntax

`xmyLoadPkg` *`packagen`*`ame ?-relocateNow? ?-initFunc` *`initFunc?`*

Return

No result

Description

The **xmyLoadPkg** command allows MYNAH and non-MYNAH extension packages to be loaded into a MYNAH SE. For standard MYNAH packages, only the (case-sensitive) logical package name must be specified. Table 6-2 lists the MYNAH package names that can be loaded.

Table 6-2. Loadable MYNAH Packages

Name
TermAsync
Term3270
TOP
PRT3270
ScriptExec
AppApp

It is assumed that MYNAH packages are in *$XMYDIR/lib* and have a package initialization function with the following name:

`xmy<`*`logical package name`*`>PkgAppInit`

for example

```
xmyTerm3270PkgAppInit
```

To load a non-MYNAH extension package using **xmyLoadPkg**, it must exist in a shared library with a global initialization function with the following syntax:

```
int AppInit(Tcl_Interp *interp);
```

This initialization function should return TCL_OK on success, or TCL_ERROR otherwise.

xmyLoadPkg takes the following options:

-relocateNow Causes all symbols to be resolved at load time, rather than when they are first accessed. This is useful when developing non-MYNAH packages.

-initFunc ***initFunc*** Specifies the name of the initialization function

> **WARNING —** The initialization function for non-MYNAH packages should not contain the prefix "xmy", since this is reserved for MYNAH packages.

There is no default search path for non-MYNAH packages, so the full path to the shared library should be specified as the ***packagename*** argument.

Subsequent calls to **xmyLoadPkg** for a package that is already loaded does nothing.

Subsequent calls to **xmyLoadPkg** for a non-MYNAH package result in the **AppInit** function being called again.

Example

```
> xmyLoadPkg TermAsync
```

Exceptions

- Missing/invalid arguments
- Package not found
- InitFunc (default or explicit) not found

6.2.10 xmyMask

xmyMask is a class command that is used to produce an instance of a mask object. A mask is used to denote some pattern of text that should be ignored during a **compare** execution. This pattern is a regular expression as defined for the **regex(3X)** program.

You may be working with an application that has screens containing dates, times, and sequence numbers. It becomes a problem during a **compare** execution when these dates, times, or sequence numbers appear within the regions to be compared. **xmyMask** provides a way of automtically masking out these dates, times, and sequence numbers.

> **NOTE —** For more information on regular expressions, see the **regex(3X)** *Manual*.

Masks can be applied to any domain level compare commands that involve text regions. Sections 6.2.10.1 through 6.2.10.4 describe the methods used with the **xmyMask** command.

6.2.10.1 create

Syntax

```
xmyMask create -pattern pattern
```

Return

Handle to a mask instance

Description

The **create** method is used to create an instance of a mask object. The callback for **create** constructs an instance of the mask object class.

create takes the following option:

-pattern ***pattern***	Specifies the regular expression ***pattern*** to be associated with the mask. When applied to an existing mask, **-pattern** becomes read-only (i.e., ***pattern*** is an invalid argument) and returns the current value.

Example

This creates a mask that masks out dates that appear in the form ***mm/dd/yy***.

```
> set dates [xmyMask create \
           -pattern {[0-1][0-9]/[0-3][0-9]/[0-9][0-9]}]
.xmyMask_1

> $dates -pattern
[0-1][0-9]/[0-3][0-9]/[0-9][0-9]
```

This one creates a mask to mask out times that appear in the form ***hh:mm:ss***.

```
> set times [xmyMask create \
           -pattern {[0-1][0-9]:[0-5][0-9]:[0-5][0-9]}]
.xmyMask_2
```

You can get specific with your masks. For example, this would mask out dates that occur during the month of July during the 1990s, asssuming the date appears in the form ***JUL dd, 199y***:

```
> set JUL [xmyMask create \
           -pattern {JUL [0-3][0-9], 199[0-9]}]
.xmyMask_3
```

6.2.10.2 destroy

Syntax

handle `destroy`

Return

No result

Description

The **destroy** method is used to destroy an instance of a mask object. Once the object has been destroyed its handle is removed from the Tcl name space. The callback for **destroy** destroys an instance of the mask object class.

Example

```
> $mask1 destroy
```

6.2.10.3 disable

Syntax

handle disable

Return

No result

Description

The **disable** method deactivates a previously enabled mask. After **disable** is called, no **compare** commands will recognize the pattern associated to the mask instance.

Example

```
> $mask disable
```

6.2.10.4 enable

Syntax

handle enable

Return

No result

Description

The **enable** method activates the mask instance so that all subsequent comparison commands within the script will ignore the pattern defined in the mask.

> **NOTE —** A new mask is not enabled automatically at creation time. See Section 6.2.10.1.

Example

```
> $mask1 enable
```

6.2.11 xmyPrint

Syntax

`xmyPrint ?-pkg` *`pkg`*`? ?-type` *`type`*`? ?-text` *`text`*`?`

Return

No result

Description

The **xmyPrint** command allows you to write to the *output* file in the script output directory. Lines are prepended with a time stamp and the category **user**. All fields are colon-separated.

xmyPrint takes the following options:

-pkg *pkg* Specifies the MYNAH extension package for which you want to create output.

-type *type* Specifies a keyword used to identify the type of output to create.

-text *text* Specifies the text to write to the *output* file.

Example

```
> xmyPrint -type “problem” -text “script shouldn’t be here”
```

would produce a line similar to the following in the *output* file:

```
19951208:140000:user::problem:script shouldn’t be here
```

Note the double colons delimiting the unspecified **pkg** field.

Exceptions

- Missing/invalid arguments
- Print operation failure.

6.2.12 xmyPrompt

Syntax

```
xmyPrompt [list {keylset_list}]
```

Return

Input provided by user as a list of strings

Description

The **xmyPrompt** command allows a script to receive data from you at script execution time. It takes a list of keyed lists as an argument. (See Appendix B.11 for a description of keyed lists.) Each sublist describes one line of the prompt window that is displayed. This command is only useful in embedded SEs, since background and standalone SEs do not have GUIs. If **xmyPrompt** is executed in a background SE, it returns an empty list.

The ***keylset_list*** items are created using the TclX **keylset** command (see Appendix B.11.4), which has the syntax

```
keylset listvar key value ?key2 value2 ...?
```

xmyPrompt uses the following **keylset** options:

-prompt ***promptString***	Specifies the prompt to display in the prompt window.
-default ***defaultValue***	Specifies a default data to enter at the prompt.
-echo ***echoFlag***	Specifies whether the entered response to the prompt will be echoed back to the prompt window. The default is TRUE.

> **NOTE —** A keyed list passed to **xmyPrompt** can contain any number of other tag-value pairs, but any pairs besides the ones listed above will be ignored by **xmyPrompt**. However, you can enter the prompt data into larger keyed lists that they are using for other reasons.

Example

```
> keylset a -prompt logid -default wunn
> keylset c -prompt password -echo false
> keylset d -prompt “other logid” -default madmin

> set result [xmyPrompt [list $a $c $d]]
```

Exceptions

Missing/invalid arguments

6.2.13 xmyReadGrep

Syntax

```
xmyReadGrep -fd fd -selection selection ?-var variable?
```

Return

Line from file if ***variable*** is not specified.

Number of characters read if ***variable*** is specified.

Description

The **xmyReadGrep** command reads the next line from ***fd*** that matches ***selection*** (a regular expression), and discards the line's terminating newline.

xmyReadGrep takes the following options:

-fd *fd*
Specifies the file to search for the specified ***selection***. This must be a ***fileid*** that was created when you opened a file using the Tcl **open** command (Appendix A.39).

Note — You can save the ***fileid*** to a variable and use this variable to specify the file.

-selection *selection*
Specifies the ***selection*** to search for in the specified field.

- If **-var *variable*** is specified, the matching line is placed in variable and the return value is the number of characters on the matching line;
- If no line matches, variable is set to the empty string and -1 is returned;
- If **-var *variable*** is not specified then the matching line (or the NULL string if no line matches) is returned.

-var variable
Specifies a variable to contain the matched text. The matching line (or Null string, if no line matches) is placed in ***variable*** and its length (or -1 if no line matches) is returned.

Each application on a particular file begins reading at the line after the last line returned.

By default, each **xmyReadGrep** application on a particular file begins reading at the line after the last line returned. Thus, if your script requires multiple **xmyReadGrep** commands, you must determine how you want your script to resume searching.

- If you want your script to search from the beginning of the file, perform one of the following:
  - — Close the file after an **xmyReadGrep** application and re-open the file before the next **xmyReadGrep** application
  - — Rewind the open file using the Tcl **seek** command.

    NOTE — The **seek** method is faster than the opening and closing files method.
- If you want your script to begin searching after the last line that was previously read, do not close the file until all **xmyReadGrep** applications are executed. Do not use the **seek** command between applications.

It is best to close the file before exiting the script since engines running in ***FullState*** mode do not close file descriptors at script completion time and since there is a limit on the number of open file descriptors a script can have at one time.

Example

In this example, you open the file *test.tcl*, saving the generated ***fileid*** to the variable ***open_file***, and you then use this variable as the ***fd*** argument to the **-fd** attribute.

```
> set open_file [open test.tcl]
file10
> xmyReadGrep -fd $open_file -selection OutputLevel
```

Here you check the return value of the **xmyReadGrep** command to see if a string of text was found.

NOTE — The next check is for the null return code ("") because the **-var** attribute was omitted.

```
> set file [open script1.tcl]
file11
> set retvalue [xmyReadGrep -fd $file -selection "hello world"]
11
> if {$result == ""} {
       xmyPrint -text "not found"
       return
}
```

This example performs the same function as the previous one, but also saves the results to the variable ***varname***.

> **NOTE —** The next check is for the return code -1 because the **-var** attribute was specified.

```
> set retvalue [xmyReadGrep -fd $file -selection "hello world" \
               -var varname]
11
> if {$retvalue == -1} {
       puts -text "not found"
}
```

This is an example of using the **seek** command to rewind the file.

> **NOTE —** The file is closed after the final **xmyReadGrep**.

```
> set ifd [open /home/user1/myfile]
file1
> set retvalue [xmyReadGrep -fd $ifd -selection  “hello world” \
               -var outputvar]
11
> seek $ifd 0
> set retvalue [xmyReadGrep -fd $ifd -selection  “.ello \[wW\]orld” \
               -var outputvar]
11
> close $ifd
```

This is an example of using the **close** command to rewind the file.

```
> set ifd [open /home/user1/myfile]
file1
> set retvalue [xmyReadGrep -fd $ifd -selection  “hello world” \
               -var outputvar]
11
> close $ifd
> set ifd [open /home/user1/myfile]
file1
> set retvalue [xmyReadGrep -fd $ifd -selection “.ello \[wW\]orld” \
               -var outputvar]
11
```

Exceptions

Missing/invalid arguments

6.2.14 xmyRegex

Syntax

```
xmyRegex -regExp regExp -data data ?-offset offset?
           ?-location location? ?-length length?
           ?-label label? ?-warning?
```

Return

1 if regular expression found

0 otherwise

Description

The **xmyRegex** command searches for a regular expression in the given input ***data*** string at ***offset*** characters into ***data***. If the regular expression is found, the offset from the beginning of the data is returned in ***location*** and the length is returned in ***length***.

xmyRegex updates the script **compare** variables (**xmyVar(GoodCompares)**, **xmyVar(FailedCompares)**, and **xmyVar(WarningCompares)**) depending on whether it finds the regular expression in the input data

xmyRegex takes the following options

-regExp *regExp*	Specifies the regular expression to search for.
-data *data*	Specifies the data string to search.
-offset *offset*	Specifies the offset into the data string in which to search.
-location *location*	Specifies a variable to contain the location of the offset from the beginning of the data where the regular expression was found.
-length *length*	Specifies a variable for the length of the matched expression.
-label *label*	Specifies a label that will appear in the *output* file's compare event.
-warning	Specifies that a non-match should increment **xmyVar(WarningCompares)** instead of **xmyVar(FailedCompares)**.

NOTE — For the -location and -length options if there is no match, the -location and -length variables are not touched. If these variables are unset before the call to xmyRegex, then they will not exist in the Tcl interpreter after the call to xmyRegex if there was no match. If you then try to print them, the script will exit with an error code. You should put the xmyRegex command in an if

statement (see example) or set the length and location variables to, say, the null string.

Side Effects

xmyRegex writes results to the *compares* file and writes a **compare** event to the *output* file.

Example

```
> if {[xmyRegex -regExp “searchstring*” -data $inputstring \
      -offset 10 -location loc -length len]} {
      puts “found at location $loc”
      puts “with length $len”
}
```

Exceptions

Missing/invalid arguments

6.2.15 xmySimilar

Syntax

```
xmySimilar -value1 value1 -value2 value2 \
           -percentage percentage
```

Return

1 if values differ by no more than the percentage specified

0 otherwise

Description

The **xmySimilar** command compares numbers to see if they differ by less than a given percentage. It does not update **xmyVar(GoodCompares)** or **xmyVar(FailedCompares)**.

> **NOTE —** The percentage used by the algorithm is the specified percent of the average of the two values supplied by the user, and hence it does not matter what order the values are given in.

xmySimilar takes the following options:

-value1 ***value1***	Specifies the first number to compare.
-value2 ***value2***	Specifies the number to compare with ***value1***
-percentage ***percentage***	Specifies the percentage by which ***value1*** and ***value2*** are expected to differ.

Example

```
> xmySimilar -value1 $result1 -value2 $result2 -percentage 5
```

Exceptions

Missing/invalid arguments

6.2.16 xmySleep

Syntax

`xmySleep` *`seconds`*

Return

No result

Description

The **xmySleep** command pauses the script for the indicated number of seconds. Data from the SUT and Telexel messages are received by the SE while **xmySleep** is active.

> **NOTE —** If a value for “seconds” is given that is too large, MYNAH will use the largest value possible under the current operating system. What this number is will depend on which hardware and operating system MYNAH is being run on.

Example

```
> xmySleep 10
```

Exceptions

Missing/invalid argument.

6.2.17 xmySymTblDel

Syntax

`xmySymTblDel -tag` *`variable`*

Returns

0 if the tag was not in the symbol table, 1 otherwise

Description

The **xmySymTblDel** command removes the tag/value pair referenced by **-tag *variable*** from the symbol table. Symbol table changes (including deletions) in a child script are propagated to the parent script when the child script finishes.

Exception

Missing/invalid argument

Example

```
> if {[xmySymTblDel -tag logid]} {
   xmyPrint -text “tag $logid found and deleted”
} else {
   xmyPrint -text “tag $logid does not exist in the symbol table”
}
```

6.2.18 xmySymTblExists

Syntax

```
xmySymTblExists -tag variable
```

Return

1 if the named variable exists in the MYNAH symbol table

0 otherwise

Description

The **xmySymTblExists** command determines if a variable, specified by **-tag *variable***, exists in the MYNAH symbol table. Since a variable can have a null value, calling **xmySymTblGet** does not determine whether the variable exists.

Example

The following example reuses the example system table we used for **xmyVar(SymTbl)** (Section 5.17). Since the tag **Conn** exists in the table, **xmySymTblExists** returns a 1, but since the tag **login** does not exist, a 0 is returned.

> **NOTE —** Remember, tag/value pairs are entered using **xmySymTblPut** (Section 6.2.20).

```
> set Conn [xmyTerm3270 connect]
.xmyTerm3270_1
> xmySymTblPut -tag Conn -value $Conn
> set xmyVar(SymTbl)
> xmySymTblPut -tag test -value 42
> set xmyVar(SymTbl)
{Conn .xmyTerm3270_1} {test 42}

> xmySymTblExists -tag Conn
1
> xmySymTblExists -tag login
0
```

6.2.19 xmySymTblGet

Syntax

```
xmySymTblGet -tag variable
```

Return

Value of ***variable*** or " " if not found.

Description

The **xmySymTblGet** command retrieves the value of ***variable*** from the symbol table. Since variables can have null values, there is no indication that an invalid variable name was specified.

You can use **xmySymTblExists** to test if a variable exists.

Example

```
> xmySymTblGet -tag Conn
.xmyTerm3270_1
```

Exceptions

Missing/invalid argument

6.2.20 xmySymTblPut

Syntax

```
xmySymTblPut -tag variable ?-value value?
```

Return

No result

Description

The **xmySymTblPut** command lets you add or change values in the symbol table and takes the following options:

-tag *variable*	Specifies the name of the tag in the symbol table to added or changed.
-value *value*	Specifies the value of the tag in the symbol table to added or changed.

Putting a value into the symbol table is equivalent to exporting a variable in versions of MYNAH prior to 5.0. The null string (“) is the default value if the value is not supplied.

Example

```
> set Conn [xmyTerm3270 connect]
.xmyTerm3270_1
> xmySymTblPut -tag Conn -value $Conn
> set xmyVar(SymTbl)
> xmySymTblPut -tag test -value 42
> set xmyVar(SymTbl)
{Conn .xmyTerm3270_1} {test 42}

> xmySymTblExists -tag Conn
1
> xmySymTblExists -tag login
0

> xmySymTblPut -tag login -value 22
> xmySymTblPut -tag hello -value 3
> xmySymTblPut -tag test -value 42
```

Exceptions

- Missing/invalid argument
- Insert Failure

6.2.21 xmyUnloadPkg

Syntax

`xmyUnloadPkg` *`packagename`*

Return

No result

Description

The **xmyUnloadPkg** command removes the named extension package from the SE. Using **xmyUnloadPkg** reduces the size of the Tcl name space slightly. In most situations it will be unnecessary. **xmyUnloadPkg** deletes the underlying package if the name matches one of the loaded MYNAH packages.

Example

```
> xmyUnloadPkg TermAsync
```

Exceptions

- Missing/invalid argument
- Package not loaded

6.2.22 xmyUpdateResult

Syntax

```
xmyUpdateResult ?-testid testId? -status string
```

Return

No result

Description

The **xmyUpdateResult** command lets you update the status for the Result object.**xmyUpdateResult** takes the following options:

-testID *testID*	Specifies the ***testid*** to use. If ***testid*** is not specified, the "current" ***testid*** is used. The "current" ***testid*** is contained in the closest enclosing **xmyBegin** command.
-status *string*	Specifies a status used to override the default status provided by the SE, which is based solely on counts of bad and warning **compare** statements. The ***string*** must be "unsuccessful", "inconclusive" or "sucessful".

Example

```
> xmyBegin -testid 200
> if { $xmyVar(failedCompares) > 17 } {
      xmyUpdateResult -status "Unsuccessful"
      xmyEnd -testid 200
      exit "max failedCompares"
}
> xmyEnd -testid 200
```

Exceptions

- Missing/invalid arguments
- No testid specified outside of an **xmyBegin**/**xmyEnd** block

6.3 Encryption Utilities

The MYNAH System includes several utilities that are used to work with encrypted files. These are the **xmyUdb** command and the CLUI's **xmyCmd scramble** sub-command.

6.3.1 xmyUdb

Syntax

```
xmyUdb read -file filename ?-decrypt ?-key key?
xmyUdb write -list list -file filename \
           ?-encrypt ?-key key?
```

Returns

xmyUdb read returns a keyed list representing the contents of file ***filename***.

xmyUdb write returns nothing.

Description

The first form of the **xmyUdb** language command reads a user database from a flat file and places it into a keyed list. The subcommand **read** must precede the other arguments. This form takes the following arguments:

-file *filename*	Contains the name of the file to be read. If just a file name is given, it is assumed to be in the SE's run directory ($XMYHOME/run/se), otherwise the user must give a full path (starting with '/'). This argument is required.
-decrypt	Instructs **xmyUdb** to decode the file before reading it.
-key *key*	Specifies the (scrambled or unscrambled) **key** to be used to decode the file before reading it. The xmyUdb read command uses the key length to determine whether it is scrambled and unscrambles it automatically before using it. If **-decrypt** is specified but a key is not provided, xmyUdb attempts to retrieve a key from the MYNAH configuration file **Engine** option **Key .**

The second form of **xmyUdb** writes a user database to the specified file. This form takes the following arguments:

-file *filename*	Contains the name of the file to which you want to write the database. If just a file name is given, it is assumed to be in the SE's run directory ($XMYHOME/run/se), otherwise the user must give a full path. This argument is required.

-list *list*	Specifies the keyed list to be written. The argument must be the name of a variable, without the "$" before it. This argument is required.
-encrypt	Causes the file to be encoded before being written.
-key *key*	Specifies the (scrambled or unscrambled) ***key*** to be used to encode the file before writing it. The xmyUdb write command uses the key length to determine whether it is scrambled and unscrambles it automatically before using it. If **-decrypt** is specified but a key is not provided, xmyUdb attempts to retrieve a key from the MYNAH configuration file **Engine** option **Key**.

The unencrypted form of the file created by **xmyUdb** write and read by **xmyUdb** contains lines of ASCII data, where each line contains a tag and a value separated by space or tab characters.

NOTE — -key can only be specified with **-decrypt** or **-encrypt**. It can not be specified separately.

Example

```
> set pwords [xmyUdb read -file $env(XMYHOME)/lib/pwords \
           -decrypt]

> set firstPassword [keylget pwords firstPassword]

> xmyUdb write -list $pwords \
           -file $env(XMYHOME)/lib/passwords \
           -encrypt
```

Exceptions

- Missing/invalid argument
- File not found
- Encryption/decryption failed
- Key not found

6.3.2 xmyCmd scramble

Syntax

```
xmyCmd scramble ?-k key? ?output-file?
```

Description

The **scramble** CLUI sub-command lets you encrypt the key associated with a script. The scrambled keys can only be unscrambled by an SE.

scramble takes the following options:

-k *key*	Specifies the ***key*** to be saved. A key can only have a maximum of eight characters. If this option is not used, you are prompted for the key.
output-file	Specifies the name of the file to which the scrambled key is to be written. If this option is not used, the scrambled key is written to the standard output.

Examples

In this example, you specify the key, and **xmyCmd scramble** generates the scrambled key.

```
> xmyCmd scramble -k desKey
f;;.W>+tv
```

This time you do not specify the key, and **xmyCmd scramble** prompts you to enter the key.

```
> xmyCmd scramble
(No need to precede special characters with a backslash)
Enter key (8 char max): cryptKey
h<.(2/Y@-
```

6.4 Performance Measurement Functions

There are no specific Tcl language commands for measuring performance. Instead, SUT timing events in the script output file and the **xmyPrint** command are used to gather timing data. For example, the following code fragment times a set of operations:

```
> set startTime [xmyDate]
> $conn send pf1       # send keys to the SUT
> $conn wait
> $conn send pf3
> $conn wait
> xmyPrint -type “timing” -text “start time: $startTime”
> xmyPrint -type “timing” -text “end time: [expr\
     [xmyDate]-$startTime]”
```

7. Child Script Extension Package

This section describes the set of Tcl extensions for the Child Script Package, which is used to start and control script execution from a parent script, implementing requests for script execution on remote SEs.

> **NOTE —** To access the Child Script Package you must first run **xmyLoadPkg ScriptExec**.

xmySE is the class command used to create connections to the BEE. Table 7-1 lists the methods used by **xmySE** to work with connections.

Table 7-1. xmySE (Child Script) Methods

Method	Description	Section
connect	Creates a connection to the background execution environment.	7.1, Page 7–2
cancel	Cancels the asynchronous request.	7.1.1, Page 7–3
destroy	Deletes the associated message object.	7.1.2, Page 7–4
pause	Temporarily disables the request associated with the message object.	7.1.3, Page 7–5
resume	Restarts a paused child script.	7.1.4, Page 7–6
send	Sends a child script execution request.	7.1.5, Page 7–7
sendWait	Sends a request for child script execution and waits for a response.	7.1.6, Page 7–8
wait	Waits until a reply is received or a timeout occurres.	7.1.7, Page 7–9

xmySE, when used with either the **waitAll** (Section 7.2) or the**waitAny** (Section 7.3) method, is also used to freeze the parent script until the appropriate response is received.

7.1 connect

Syntax

```
xmySE connect ?-sd sdName? ?-se seGroupName?
                  ?-timeout timeout?
```

Return

A handle to the connection used by subsequent **send** and **sendWait** methods.

Description

The **xmySE connect** method creates a connection to the BEE. **connect** optionally takes the name of the SD and the SE group to which the parent script wants to send requests.

connect accepts the following options:

-sd *sdName*	Specifies the name of the SD to use when creating a connection. If this is not specified, **connect** uses the default from the MYNAH *xmyConfig* file. This option is only useful when its package is being used by the GUI process, since an SE cannot send to other SDs.
-se *seGroupName*	Specifies the name of the SE group to use when creating a connection. If this is not specified and you are running **connect** in the Script Builder, the environment variable XMYSEGROUP must be set to the desired SE Group, e.g., XMYSEGROUP=SeGp1.
-timeout *timeout*	Determines how long future **wait** and **sendWait** commands will wait for the child script to complete executing before aborting. If you do not specify a ***timeout***, this option will default to 30 minutes.

Example

```
> set hSE [xmySE connect -timeout 10]
.xmySE_1
```

Exceptions

- Invalid/missing arguments
- Telexel connect failure

7.1.1 cancel

Syntax

msgHandle cancel

Return

No result

Description

The **cancel** method cancels the asynchronous request associated with the message object handle. On return, the message object is deleted.

Example

```
> $hSE cancel
```

7.1.2 destroy

Syntax

msgHandle destroy

Return

No result

Description

The **destroy** method deletes the associated message object. It does not send a request to the child script.

> **NOTE —** Destroy should only be used for message objects created using the send command. It should not be used after sendWait command as sendWait does not create any message object.

Example

```
> $hSE destroy
```

7.1.3 pause

Syntax

msgHandle pause

Return

No result

Description

The **pause** method pauses a child script. While the child script is paused the parent continues to execute. It is not an error to call **pause** more than once for the same script.

Example

```
> $hSE pause
```

7.1.4 resume

Syntax

msgHandle resume

Return

No result

Description

The **resume** method restarts a paused child script. It is not an error to call **resume** more than once for the same script.

Example

```
> $hSE resume
```

7.1.5 send

Syntax

handle send -script *scriptName*

Return

A handle to a message object used to identify the outstanding request.

Description

The **send** method sends a child script execution request. The current symbol table is passed with the request.

send accepts the following option:

-script *scriptName*	Specifies the name of the child script to execute. The **scriptName** can be a full path, a relative path, or a base file name. Relative paths, such as *./script1.tcl or ../../script1.tcl*, are assumed to be relative to the paths specified in the configuration option ***TclLibraryPath***. Base files names are also assumed to be in one of the ***TclLibraryPath*** directories. Specifying a ***scriptName*** starting with "/" turns off ***TclLibraryPath*** searching.

The **send** command returns immediately.

Exceptions

- Invalid/missing arguments
- Telexel Send operation failure
- Timeout

7.1.6 sendWait

Syntax

```
handle sendWait -script scriptName
           ?-timeout timeout?
```

Return

The exit string of the child script

Description

This **sendWait** method sends a request for child script execution and waits for a response, which is a message indicating that the child script has completed executing. The response contains any new or changed symbol table variables. **sendWait** updates the parent script symbol table before returning.

sendwait accepts the following options:

-script *scriptName*	Specifies the name of the child script to execute. The **scriptName** can be a full path, a relative path, or a base file name. Relative paths, such as *./script1.tcl or ../../script1.tcl*, are assumed to be relative to the paths specified in the configuration ***LibraryPath*** parameter. Base files names are also assumed to be in one of the ***LibraryPath*** directories. Specifying a ***scriptName*** starting with "`/`" turns off ***LibraryPath*** searching.
-timeout *timeout*	Specifies the amount of time to wait for a response before aborting.

The **send** and **sendWait** methods compose and send a request to the SD.

The symbol table passed to the child script is the same as the current symbol table in the parent script.

Exceptions

- Invalid/missing arguments
- Telexel Send failure
- Timeout

Side Effects

The symbol table is updated with new and changed values from the child script.

7.1.7 wait

Syntax

msgHandle wait ?-timeout *timeout*?

Return

The exit string of the child script.

Description

The **wait** method waits until timeout is reached or a reply to the outstanding **send** is received, which is a message indicating that the child script has completed executing. ***handle*** is a message object identifier.

Timeout specifies the maximum amount of time to wait for a response.

Exceptions

- Missing/invalid arguments
- Timeout

Side Effects

The symbol table is updated with new and changed values from the child script.

7.2 xmySE waitAll

Syntax

```
xmySE waitAll ?-messages {msgHandle1 ... handleN}?\
                ?-timeout timeout?
```

Return

No return

Description

The **xmySE waitAll** method blocks the parent script until all of the requested child scripts complete or the timeout expires.

xmySE waitAll accepts the following options:

-messages *{handle1 ... handleN}*	Specifies the handle(s) associated with a script(s). The argument to **-messages** is a Tcl list of message handles. If **-messages** is not specified, **waitAll** blocks until replies are received for all child script execution requests.
-timeout *timeout*	Specifies the amount of time to wait for a response before aborting. The default is 1800 seconds.

For each specified message, script compare counts and the symbol table are only updated the first time the reply to the send request is seen. A reply is "seen" when it is waited on. Therefore it is safe to wait on the same message more than once. For example, after **waitAll** returns, the **wait** method can be used to retrieve the Tcl interpreter result for a specific send request.

Exceptions

- Nonexistant message handle specified
- No send requests to wait for
- Timeout
- Script cancelled while waiting

7.3 xmySE waitAny

Syntax

```
xmySE waitAny ?-messages {msgHandle1 ... handleN}?\
                ?-timeout timeout?
```

Return

The message handle associated with the script that completed

Description

The **xmySE waitAny** method blocks the parent script until one of the requested child scripts completes or the timeout expires.

xmySE waitAny accepts the following options:

-messages *{handle1 ... handleN}*	Specifies the handle(s) associated with a script(s). The argument to **-messages** is a Tcl list of message handles. If **-messages** is not specified, **waitAny** defaults to any child script request (even requests that have already been waited on.)
-timeout *timeout*	Specifies the amount of time to wait for a response before aborting. The default is 1800 seconds.

For each specified message, script compare counts and the symbol table are only updated the first time the reply to the send request is seen. A reply is "seen" when it is waited on. Therefore it is safe to wait on the same message more than once.

For example, after **waitAny** returns, the **wait** method can be used to retrieve the Tcl interpreter result for the child script that finished.

Each invocation of **waitAny** is completely independent, meaning that multiple invocations of **waitAny** with the same message list will each return the same message handle.

Exceptions

- Nonexistant message handle specified
- No send requests to wait for
- Timeout
- Script cancelled while waiting
- Execution results in FMM reply improperly formatted
- Execution results missing data

8. TermAsync Extension Package

8.1 Overview

The TermAsync Extension Package provides the functions necessary for interactions with the asynchronous terminal device. This package provides emulation of the vt100 terminal as well as partial emulation of other terminals, based upon the system terminfo database.

> **NOTE —** To access the TermAsync Extension Package you must first run **xmyLoadPkg TermAsync**.

The following is the list of commands in the MYNAH TermAsync Extension Package.

8.1.1 Methods Overview

Section 8.4.1 contains detailed descriptions of the TermAsync Method extensions. The extensions are listed in alphabetical order (within each category). Table 8-1 lists the extensions, organizing them in general functional categories. Table 8-1 also gives a brief description of each extension and the section where the detailed description can be found.

Table 8-1. TermAsync Method Extensions (Sheet 1 of 2)

Category	Method	Description	Section
Connection	**connect**	Establishes a connection to the asynchronous host from the MYNAH Tcl script.	8.4.1.2, Page 8–8
	disconnect	Destroys a connection made to the host through the **connect** method.	8.4.1.4, Page 8–11
Data Entry/Retrieval	**response**	Returns the latest response of the application.	8.4.1.8, Page 8–15
	send	Sends a string to the SUT.	8.4.1.10, Page 8–19
	screen	Returns a list of strings that represents a screen image.	8.4.1.9, Page 8–17
Comparisons	**compare**	Compares a region of an asynchronous screen with a compare pattern body.	8.4.1.1, Page 8–6
	disableMask	Disables a mask object.	8.4.1.3, Page 8–10
	enableMask	Enables an already created mask object for a particular connection instance.	8.4.1.5, Page 8–12

Table 8-1. TermAsync Method Extensions (Sheet 2 of 2)

Category	Method	Description	Section
Waiting	**sendWait**	Sends a string to the SUT and waits until a second string is returned from the SUT.	8.4.1.11, Page 8–21
	wait	Stops script execution until a condition is verified or a timeout occurs.	8.4.1.12, Page 8–22
Attributes	**getAttributes**	Returns the attribute (e.g., blinking, highlighted, etc.) value at specified position.	8.4.1.6, Page 8–13
	listAttributeTypes	Returns the list of valid attributes.	8.4.1.7, Page 8–14

8.1.2 Attributes Overview

Section 8.4.2 contains detailed descriptions of the TermAsync Attribute extensions. The extensions are listed in alphabetical order. Table 8-2 lists the extensions, organizing them in general functional categories. Table 8-2 also gives a brief description of each extension and the section where the detailed description can be found.

Table 8-2. TermAsync Attribute Extensions (Sheet 1 of 2)

Category	Method	Description	Section
Connection	**-connections**	Returns a list of asynchronous connections.	8.4.2.3, Page 8–25
	-name	Returns the name of the connection.	8.4.2.8, Page 8–26
	-shell	Returns the start-up shell.	8.4.2.12, Page 8–27
	-status	Returns the status of the connection.	8.4.2.15, Page 8–28
	-terminal	Returns the terminal being emulated.	8.4.2.16, Page 8–29
	-terminfo	Returns the file name of the auxiliary *terminfo* file.	8.4.2.17, Page 8–29

Table 8-2. TermAsync Attribute Extensions (Sheet 2 of 2)

Category	Method	Description	Section
Data Entry/Retrieval	**-bufferlen**	Returns the size of the buffer that caches SUT responses.	8.4.2.1, Page 8–24
	-delay	Returns or sets the number of milliseconds used as padding in the **send** statement.	8.4.2.4, Page 8–25
	-size	Returns the size of the screen.	8.4.2.14, Page 8–28
Location	**-column**	Returns the cursor's current column position.	8.4.2.2, Page 8–24
	-position	Returns the row and column position of the cursor.	8.4.2.9, Page 8–27
	-row	Returns the cursor's current row position.	8.4.2.11, Page 8–27
Comparisons	**-failedCompares**	Returns or sets the number of failed compares.	8.4.2.5, Page 8–25
	-goodCompares	Returns or sets the number of successful compares.	8.4.2.6, Page 8–26
	-masks	Returns a list of enabled masks for a connection	8.4.2.7, Page 8–26
	-warningCompares	Returns or sets the number of user defined warnings.	8.4.2.19, Page 8–29
	-wildcard	Returns or sets the default comparison wildcard character.	8.4.2.20, Page 8–30
Waiting	**-prompt**	Returns or sets the default waiting string.	8.4.2.10, Page 8–27
	-timeout	Returns or sets the default number of seconds to wait for a SUT response.	8.4.2.18, Page 8–29
Attributes	**-showAttributes**	Returns or sets what character attributes are to be included in the *SUTimage* file.	8.4.2.13, Page 8–28

8.2 System Prompts

UNIX prompts often differ from installation to installation. What prompts are expected also differ if you execute scripts in the background or the foreground.

> **NOTE —** This section uses the TermAsync method **sendWait** to illustrate these techniques. Briefly, **sendWait** sends a string, such as a login id or a UNIX command, to the connection and waits until an expected string is returned, such as
>
> ```
> sendWait "ls\r" -expect "\$ "
> ```
>
> In this case, **sendWait** send the **ls** command and waits until it receives the $ prompt. For a complete explanation of **sendWait** see Section 8.4.1.11.

As you use the Script Builder to capture an emulated connection, it automatically generates a **sendWait** statement each time you enter a command, such as **ls** or **ftp**, entering the prompt sign it encounters as the expected string. The Script Builder determines the prompt to be the shortest sub-string from the end of the response that is unique (i.e., only one occurrence in the response). An example would be the prompt for an ftp session, `ftp>`, for which **sendWait** would enter `"> "` as the expected string, ignoring the system id.

If you then run the script using the Script Builder, this is no problem since you created the script in the foreground and are executing it in the foreground; the same prompt is expected. If you execute the script in the background, however, this can cause problems since the system may assumes the local prompt sign is the UNIX default prompt, the dollar sign ($). If your local prompt sign is different, your script will fail since it is waiting for a response it will not receive.

You can simply change all expected strings for a local connection to the dollar prompt. If you use the script to connect to a remote system (such as via **telnet** or **ftp**), you will not have to do this; the system will expect the prompt the Script Builder generated.

There is always the possibility, however, that there may be discrepancies between the expected prompt and the actual prompt. If you want to be completely sure that the prompts will match, you can use your script to specify the prompt by adding a line similar to

```
sendWait “PS1=\”$ \”\r” -expect “\$ “
```

immediately after the TermAsync **connect** method. (See Section 8.4.1.2 for information on the **connect** method).

The following script automates an anonymous **ftp** session. The lines up to the line where we send the **ftp** command are all on the local system. To ensure that the actual and expected prompt match, we add the above line to export the `PS1` variable. The other lines (up to the **bye**, which closes the **ftp** connection) all expect the prompt you would encounter during an **ftp** session; they would not expect the prompt we exported.

```
xmyLoadPkg TermAsync
set conn2 [xmyTermAsync connect]
$conn2 wait {[$conn2 response -numberOfCharacters] >= 1}
$conn2 sendWait “PS1=\”\$ \”\r” -expect “\$ “
$conn2 wait -expect “\$ “
$conn2 sendWait “cd /users/kjb/NEW\r” -expect “\$ “
$conn2 sendWait “ftp fake_ftp.com\r” -expect “: “
$conn2 sendWait “anonymous\r” -expect “d:”
$conn2 sendWait “kjb@\r” -expect “> “
$conn2 sendWait “cd pub/new_this_week\r” -expect “> “
$conn2 sendWait “prompt\r” -expect “> “
$conn2 sendWait “mget *\r” -expect “> “
$conn2 sendWait “bye\r” -expect “\$ “
$conn2 disconnect
xmyExit
```

Where you enter the line to change the PS1 variable is very important. When the Script Builder creates a connection, it immediately generates a **wait** statement as in

```
set conn2 [xmyTermAsync connect]
$conn2 wait -expect “: “
```

You must place the line to change the PS1 variable between these two lines, and edit the expected value of the **wait** statement.

8.3 Waiting for a Response

The TermAsync package contains two methods, **sendWait** and **wait**, that suspend the script execution until the SUT has returned with the proper response. The arguments to these methods are **-timeout** and **-expect**, where:

- **timeout** is the amount of the time the **wait** should wait before timing out and calling the timeout handler. If the **timeout** is set to 0, the SE returns control back to the script.
- **expect** is the literal string to search for when data is received from the SUT. If data is available, control is returned back to the script. If no data is available, an exception is thrown (fails), and the time-out handler is called.

When no **expect** data is supplied, the **wait** will complete when the first receive is made from the SUT. Otherwise, when a **timeout** and **expect** are supplied, the **wait** command will wait until the host has transmitted the exact string specified in the **-expect** argument, or until the supplied **timeout** time has been reached, at which point the **timeout** handler will be called.

> **NOTE —** If your script has multiple **waits**, it may be best to combine them into one **wait** method with an **expect** rather than have the script pause as it processes each **wait**.

8.4 xmyTermAsync class

xmyTermAsync is the Tcl class command providing language extensions that are necessary for interactions with the asynchronous device. Its methods provide the basis of operations on an asynchronous connection that you will need to create sufficiently sophisticated test cases in your scripts.

8.4.1 Methods

The TermAsync package contains the following methods.

8.4.1.1 compare

Syntax

```
handle compare -region region -expect data \
     ?-ignore region? … ?-ignore region? \
     ?-mask maskHandle? … ?-mask maskHandle? \
     ?-wildcard char? -warning \
     ?-outputLabel label?
```

Return

1 if the data on screen matches the expected data

0 otherwise

Description

The **compare** method compares a region of the screen against the expected data, entered using the **-expect *data*** option. If a comparison is successful, the count of good compares for the connection is increased by one. If a comparison fails, the number of failed compares for the connection is increased, unless **-warning** is specified, in which case the number of warnings is increased. Also the global variables **GoodCompares**, **FailedCompares**, and **WarningCompares** are updated accordingly.

compare takes the following options:

-region *region*	Defines the region of the screen to compare against the expected data. ***region*** is a list of four numbers ***{row column width height}***, e.g., {1 1 80 24}.

-expect ***data***	Defines the expected content of the screen region. ***data*** is a Tcl list of lists, where each list represents a row in the region to be compared. If the actual data contains the wildcard character, the character will match any character on the screen, for that particular position.
-mask ***maskHandle***	Specifies a handle to an **xmyMask**. (See Section 6.2.10 for a discussion on **xmyMask**.) The **compare** statement ignores the subregions identified by the masks that are active, and the subregions identified by the masks that are passed as arguments.
-ignore ***region***	Defines a subregion **compare** will ignore.
-wildcard ***char***	Defines the wildcard character, the default is *****.
-warning	Increase the number of warnings rather than the number of failed compares.
-outputLabel ***label***	Specifies a label for this particular **compare** command that will be written into the output file. The label from the output file can be used to refer back to the **compare** command in the input script. This is for cross reference purposes only.

Example

```
> set main_screen(date_time_field) {1 15 17 1}
> set main_screen(user_field) {1 39 10 1}
> set my_screen {1 1 80 5}
> $myconn compare -region $my_screen -expect {\
     {MY APPLICATION Main Menu                        } \
     {Date and Time: 05/23/95 15:30:01 User: joe      } \
     {                                                } \
     {1. Screen1                                      } \
     {2. Screen2                                      } \
}
-ignore $main_screen(date_time_field)
-ignore $main_screen(user_field)
> > set t1 {“MAIN MENU”}
> $conn1 compare -tag “title” -expect $t1
```

8.4.1.2 connect

Syntax

```
xmyTermAsync connect \
     ?-terminal string? \
     ?-timeout seconds? \
     ?-shell filename? \
     ?-bufferlen length? \
     ?-terminfo filename? \
     ?-name connection_name? \
     ?-wildcard char?\
     ?-delay milliseconds?
     ?-showAttributes on|off?
     ?-size {row column}?
     ?-prompt string?
```

Return

Handle to an asynchronous connection.

Description

The **connect** method is used to establish a connection to the asynchronous host from the MYNAH Tcl script. **connect** returns a handle to a connection back to the script. Default configuration parameters are provided. The following options can be used to override the default configuration parameters:

-terminal *string*	Sets the terminal type being emulated.
-timeout *seconds*	Sets the time that MYNAH waits for a response from the SUT (time-out).
-shell *filename*	Defines the shell invoked when MYNAH connects to the SUT. **—** The default shell that is invoked when you create an asynchronous connection is the k-shell. If you don't need the k-shell, you may want to specify the bourne shell, i.e. enter **-shell */bin/sh***, since it is simpler and requires less overhead.
-bufferlen *length*	Defines the size of the buffer that stores the last response form the SUT. (It must be a non-negative number.)
-terminfo *filename*	Defines the location of the auxiliary terminfo file.
-name *connection_name*	Defines the name of the connection, overwriting the default **.xmyTermAsync_N**.
-wildcard *char*	The character used for masking regions.
-delay *milliseconds*	The delay in milliseconds used by the **send** method.

-showAttributes *on*|*off*	Determines what character attributes are to be included in the *SUTimage* file. The ***on*** option causes the character attributes on the screen to be included in the *SUTimage* file. The ***off*** option causes the character attributes on the screen to NOT be included in the *SUTimage* file.
-size {*row column*}	Sets the row and column size of the screen. The size is specified as a list in the format, ***{row column}***, where $24 \leq$ ***row*** ≤ 50 and $80 \leq$ ***column*** ≤ 132.
-prompt	Specifies a default string that the **wait** and **sendWait** methods are using if no other argument is supplied.

Example

```
> set myconn [xmyTermAsync connect -terminal vt100 \
                          -timeout 10 -size {30 90}]
```

Exceptions

- **connect** will fail (throw an exception) if the connection cannot be established (because of an Operating System error).
- **connect** will fail if a duplicate name is specified.

8.4.1.3 disableMask

Syntax

handle disableMask *list_of_masks*

Return

None

Description

The **disableMask** method disables a mask, created by the **xmyMask** command, for the connection.

Example

```
> $myconn disableMask $mask1 $mask2
```

Exception

Mask handle not enabled on connection.

8.4.1.4 disconnect

Syntax

handle `disconnect`

Return

None

Description

The **disconnect** method destroys the connection handle.

Example

```
> $myconn disconnect
```

8.4.1.5 enableMask

Syntax

handle enableMask *list_of_masks*

Return

None

Description

The **enableMask** method enables a mask, created by the **xmyMask** command, for the connection. Any subsequent **compare** statements on the connection will ignore the pattern defined in the mask object.

Example

```
> $myconn enableMask $mask1 $mask2
```

Exception

Invalid mask handle

8.4.1.6 getAttributes

Syntax

handle `getAttributes ?-position` *{row column}?*

Return

A list of attributes

Description

The **getAttributes** method returns the attribute (i.e. blinking, highlighted etc.) value at the given row and column. If the **-position** option is not specified then the current position is taken as the default.

Example

```
> $myconn getAttributes
BOLD, BLINK
```

8.4.1.7 listAttributeTypes

Syntax

```
xmyTermAsync listAttributeTypes
```

```
handle listAttributeTypes
```

Return

```
NONE, STANDOUT, UNDERLINE, REVERSE, BLINK, DIM, BOLD,
INVISIBLE, PROTECT, GRAPHIC, ALTERNATE
```

Description

The **listAttributeTypes** method returns the list of valid attributes.

Example

```
> $myconn listAttributeTypes
STANDOUT UNDERLINE REVERSE BLINK DIM BOLD INVISIBLE PROTECT
```

8.4.1.8 response

Syntax

```
handle response ?-tail number? ?-file filename? ?-append?

handle response ?-head number? ?-file filename? ?-append?

handle response -unique

handle response -numberOfCharacters

handle response -numberOfLines

handle response ?-after string? -after# integer? \
          ?-before string? ?-before# integer? \
          ?-file filename? ?-append?
```

Return

A string or an integer

Description

The **response** method returns the latest response of the application. If no arguments are specified, the complete response from the application is returned. The characters sent by the SUT between two **send** statements define the latest system response.

The response may contain the last string that was sent if the SUT does echoing (e.g., as does the UNIX shell).

response takes the following options:

-tail *number*	Returns the ***number*** of lines or characters from the end of the last system response.
-head *number*	Returns the ***number*** of lines or characters from the beginning of the last system response. For **-tail** and **-head**, ***number*** is an integer followed by the suffix **ch** or **ln** (for characters or lines, respectively).
-file *filename*	Returns an empty string and the response is saved to the specified filename.
-append	Used with **-file**, **-append** appends the response to the specified ***filename***. If this attribute is not used and ***filename*** is an existing file, **response** will overwrite the file.
-unique	Returns the shortest string from the end of the response that does not have any other occurrences in the response.
-numberOfCharacters	Returns the number of characters sent by the application in its last response.

-numberOfLines	Returns the number of lines of last application response.
-after *string*	Returns the SUT response after the first (or user specified) occurrence of the ***string*** pattern.
-after# *integer*	Specifies which occurrence after the ***string*** pattern to return.
-before *string*	Returns the SUT response after the before (or user specified) occurrence of the ***string*** pattern.
-before# *integer*	Specifies which occurrence before the ***string*** pattern to return.

NOTE — The latest response of the application is returned with all embedded null characters (i.e., ASCII 0) stripped out. The **-numberOfCharacters** option will return the actual number of characters including null characters. Null characters are not stripped if the response is written to a file (using **-file**).

Exceptions

An exception is raised if the receive fails, for example because the connection died.

Example

```
> $myconn send “ls -la\n”
> $myconn wait “myprompt $”
> set lines [$myconn response -numberOfLines]
> $myconn response -tail 10ch
```

8.4.1.9 screen

Syntax

```
handle screen ?-region region?
     ?-mask maskHandle? … ?-mask maskHandle?
     ?-ignore region? … ?-ignore region?\
     ?-wildcard char? ?-file filename? ?-append?

handle screen -score ?-method name?
```

Return

The first format of the **screen** method returns a list of lists representing a screen image. If **-file *filename*** is specified, the list is saved to a file.

The second format returns an integer.

Description

The **screen** method returns a string that represents a screen image, specified in a list format, of the ***region ({row column width height})***. Each element of the list is a list of characters (or strings) that represents a row in the screen.

screen takes the following options:

-region *region*	Defines the region parameters of the screen image to return. ***region*** is a list in the format ***{row column width height}***, e.g., a valid entry would be **-region *{1 40 40 24}***.
-mask *maskHandle*	Specifies a handle to an **xmyMask**. (See Section 6.2.10 for a discussion on **xmyMask**.) The **screen** statement ignores the sub-regions identified by the masks that are passed as arguments.
-ignore *region*	Defines a sub-region **screen** will ignore.
-wildcard *char*	Defines the wildcard character that will replace the characters in the **-ignore *region*** subregion. The default is *****.
-file *filename*	Returns an empty string and saves the response to the specified filename.
-append	Used with **-file**, **-append** appends the response to the specified ***filename***. If this attribute is not used and ***filename*** is an existing file, **response** will overwrite the file.
-score	Returns an integer representing the screen score.

-method *name* Defines the method used to compute the screen score. In the ***normal*** method, the score is computed by assigning to each character on the screen a score from 1 to 128 and adding them up. In the method is ***highlight***, the single character score is incremented by 128 if the character is highlighted. The default is ***normal***.

Example

This example captures the current screen and saves to the file *myfile*.

```
> set file [open “myfile” w]
> set myscreen {$myconn screen -region {1 1 80 25}\
     -mask {10 10 5 5} -wildcard #
> puts $file $myscreen
```

This example can be more concisely re-written as the following:

```
> $myconn screen -region {1 1 80 25} -mask {10 10 5 5}\
      -wildcard # -file “myfile”
```

8.4.1.10 send

Syntax

```
handle send string-expression ?-delay milliseconds? ?-secret?

handle send -key key ?-repeat number? ?-delay milliseconds?\
                ?-secret?
```

Return

None

Description

The **send** method evaluates ***string-expression***, sends it to the SUT, and updates the internal data representation of the screen. ***string-expression*** can be any Tcl expression that returns a string.

send also takes the following options:

-delay *milliseconds* Represents the padding. If the padding is greater than zero the system sends a character at a time, and it waits for the specified number of milliseconds before sending the next character.

-key *key* Specifies the escape sequence defined for the key that is sent to the SUT.

The following special keys are defined by default. (The escape sequence is defined in the *terminfo* or *auxiliary terminfo* files.) This argument is case insensitive.

```
backspace, return, delete, line_feed,
enter, esc, tab,

left_arrow, down_arrow, up_arrow,
right_arrow, right_1 .. right_10,

left_1 .. left_10,top_1 .. top_9,
numeric_0 .. numeric_9,

pf1.. pf4.
```

-repeat *number* Specifies the number of times the key is sent.

-secret Specifies that the string that is sent won't be recorded in the *SUTimage* file in the ***String Sent*** section. Instead, the string *<hidden data>* is placed in the file.

Note — If the application echoes the string that is sent on the screen, the *SUTimage* file will contain the string in the ***Screen*** and ***Response*** section of the file.

Example

```
> $myconn send “/usr/local/bin/dq\n”
> $myconn send -key tab -repeat 3
> $myconn send “25967”
> $myconn send -key return
```

Exceptions

An exception is raised if the **send** fails, for example because the connection died.

8.4.1.11 sendWait

Syntax

```
handle sendWait string-expression \
     ?-expect string-expression? ?-timeout timeout? ?-secret?
```

Return

The **sendWait** method sends a string to the SUT and waits until a second string is returned from the SUT. It is a shorthand for a **send** immediately followed by a **wait**.

sendWait takes the following options:

string-expression	Specifies the string to send to the SUT.
-expect *string-expression*	Specifies the string you expect to be returned by the SUT. If you don't specify an expected string, **sendWait** will expect the default prompt. This default prompt can be • Set on a per connection basis an option to the **connect** command (Section 8.4.1.2.) • Changed for the current connection using the **-prompt** attribute (Section 8.4.2.10.) • Applied to the **xmyTermAsync** class command to set a default script-wide prompt. **Note —** If the response contains null characters (i.e., ASCII 0), the nulls are stripped from the response before comparing with the expected string. Therefore, you should not have to worry about embedded null characters in the expected string.
-timeout *timeout*	Specifies number of seconds to wait before **sendWait** returns control back to the script
-secret	Specifies that the string that is sent won't be recorded in the *SUTimage* file in the ***String Sent*** section. Instead, the string <*hidden data*> is placed in the file. **Note —** If the application echoes the string that is sent on the screen, the *SUTimage* file will contain the string in the ***Screen*** and ***Response*** section of the file.

Example

```
> $myconn sendWait “ls -la\n” -expect “myprompt $”
```

8.4.1.12 wait

Syntax

handle wait *tcl-expression* ?-timeout *seconds*?

handle wait -expect *expression* ?-timeout *seconds?*

Return

No result

Description

The **wait** method stops the script execution until a particular condition is verified or a **timeout** has occurred. In its first form, **wait** stops execution until an entered Tcl expression is satisfied. In its second form, execution waits until the expected ***expression*** is encountered.

wait also takes the following options:

-timeout *seconds* Stop execution for the specified number of seconds. If a **timeout** value is not set explicitly, the default connection **timeout** value is assumed.

-expect *expression* The script waits until the string specified in ***expression*** appears at the end of the SUT response. If you don't specify an expected string, **wait** will expect the default prompt. This default prompt can be

- Set on a per connection basis an option to the **connect** command (Section 8.4.1.2.)
- Changed for the current connection using the **-prompt** attribute (Section 8.4.2.10.)
- Applied to the **xmyTermAsync** class command to set default a script-wide prompt.

Note — If the response contains null characters (i.e., ASCII 0), the nulls are stripped from the response before comparing with the expected string. Therefore, you should not have to worry about embedded null characters in the expected string.

Example

The following lines first determines what the prompt is on the system to which you created a connection. It then sends an **ls** command to the system and waits until the system returns the prompt.

```
> set myconn [xmyTermAsync connect]
> xmySleep 2; #sleep for a few seconds
> set prompt [$myconn response -tail 1ln]
> $myconn send “ls -la\n”
> $myconn wait -expect $prompt
```

If you know what the prompt will be, you can enter it directly as the argument to **-expect**, as in the following:

```
> $myconn wait -expect "prompt % "
```

This time you want execution to wait until the cursor is in row 10 and column 10.

```
> $myconn wait { [$h -position] == {10 10} }
```

Exceptions

Timeout reached, SUT has not returned, timeout handler called

8.4.2 Attributes

The following is the list of attribute methods that can be used for a particular connection to get (or set) configuration parameters or status information.

8.4.2.1 -bufferlen

Syntax

handle `-bufferlen`

Description

The **bufferlen** attribute returns the size of the buffer that caches SUT responses.

8.4.2.2 -column

Syntax

handle `-column`

Description

The **column** attribute returns the current column position.

8.4.2.3 -connections

Syntax

```
xmyTermAsync -connections
```

Return

List of asynchronous connections

Description

The **connections** attribute lists all the open asynchronous connections.

Example

```
> xmyTermAsync -connections
{.xmyTermAsync_1 xmyTermAsync_2}
```

8.4.2.4 -delay

Syntax

handle -delay ?*milliseconds*?

Description

The **delay** attribute returns or sets the number of ***milliseconds*** used as padding in the **send** method.

8.4.2.5 -failedCompares

Syntax

handle -failedCompares *?number?*

Description

The **failedCompares** attribute returns or sets the number of failed compares.

8.4.2.6 -goodCompares

Syntax

handle -goodCompares *?number?*

Description

The **goodCompares** attribute returns or sets the number of successful compares performed using the **compare** method for this handle.

8.4.2.7 -masks

Syntax

handle -masks

Return

List of enabled masks

Description

The **-masks** attribute lists all the mask handles associated to a particular connection.

Example

```
> $myconn -masks
.xmyMask01 .xmyMask03
```

8.4.2.8 -name

Syntax

handle -name

Description

The **name** attribute returns the name of the connection.

8.4.2.9 -position

Syntax

handle `-position`

Description

The **position** attribute returns the position of the cursor in a list format: ***{row column}***.

8.4.2.10 -prompt

Syntax

handle `-prompt ?`*string*`?`

Description

The **prompt** attribute returns or sets the default waiting string for the **sendWait** and **wait** methods.

8.4.2.11 -row

Syntax

handle `-row`

Description

The **row** attribute returns the current row position.

8.4.2.12 -shell

Syntax

handle `-shell`

Description

The **shell** attribute returns the shell used to start up the connection, e.g., */bin/sh.*

8.4.2.13 -showAttributes

Syntax

handle -showAttributes ?*mode?*

Description

The **showAttributes** attribute returns (**true** or **false**) or sets the mode (**on** or **off**) of the **connect** method's **-showAttributes** attribute. That is, this determines what character attributes are to be included in the *images* file.

8.4.2.14 -size

Syntax

handle -size

Description

The **size** attribute returns the size of the screen in a list format: ***{row column}***.

8.4.2.15 -status

Syntax

handle -status

Description

The **status** attribute returns the connection's status, whether it is alive(up) or not (down).

0 - The connection is down

1 - The connection is up.

8.4.2.16 -terminal

Syntax

handle `-terminal`

Description

The **terminal** attribute returns the terminal being emulated, e.g., vt100.

8.4.2.17 -terminfo

Syntax

handle `-terminfo`

Description

The **terminfo** attribute returns the file name of the auxiliary *terminfo* file.

8.4.2.18 -timeout

Syntax

handle `-timeout ?`*seconds*`?`

Description

The **timeout** attribute returns or sets the default number of seconds to wait for a SUT response.

8.4.2.19 -warningCompares

Syntax

handle `-warningCompares` *?number?*

Description

The **warningCompares** attribute returns or sets the number of user defined warnings.

8.4.2.20 -wildcard

Syntax

handle `-wildcard ?`*char*`?`

Description

The **wildcard** attribute returns or sets the default wildcard character used in **compare** and **screen** methods.

8.4.3 Changing Configuration Parameters

Syntax

```
xmyTermAsync \
     ?-terminal string? \
     ?-timeout seconds? \
     ?-shell filename? \
     ?-bufferlen length?  \
     ?-terminfo filename? \
     ?-wildcard char?\
     ?-delay milliseconds?
     ?-showAttributes mode?
     ?-prompt string?
```

Return

None

Description

All of the arguments for the **connect** method (Section 8.4.1.2), with the exception of the **-name** argument, can also be applied to the **xmyTermAsync** class command to set default attributes script-wide. After the command is issued, all the new connections will inherit the new default for the attribute.

Example

To change the default timeout to 10 seconds, type

```
> xmyTermAsync -timeout 10
```

8.4.4 Querying Configuration Parameters

Syntax

```
xmyTermAsync -terminal
xmyTermAsync -timeout
xmyTermAsync -shell
xmyTermAsync -bufferlen
xmyTermAsync -terminfo
xmyTermAsync -wildcard
xmyTermAsync -delay
xmyTermAsync -showAttributes
xmyTermAsync -prompt
```

Return

A Tcl string (or integer) that represent the current value of the parameter,

Description

All of the arguments for the **connect** method (Section 8.4.1.2), with the exception of the **-name** argument, can also be used with the **xmyTermAsync** class command to query the current default configuration parameters.

Example

```
> xmyTermAsync -terminal
vt100

> xmyTermAsync -timeout
10
```

8.5 Async Scripting

The previous section detailed the extensions in the TermAsync Package, but how do these extensions fit together to form scripts. Let's look a few examples.

In this first example (Figure 8-1) we want to

1. Logon to a remote system
2. Use the **ls** command to display the contents of the current directory.
3. Use the **pwd** command to display the current directory
4. Retrieve this information
5. Disconnect from the remote system.

```
xmyLoadPkg TermAsync
keylset c -prompt password -echo false
set a [xmyPrompt [list $c]]
set conn1 [xmyTermAsync connect]
$conn1 wait -expect ": "
$conn1 sendWait "rlogin 128.96.186.123\r" -expect "d:"
$conn1 send "$a\r"
$conn1 sendWait "ls\r" -expect "# "
$conn1 sendWait "pwd\r" -expect "# "
$conn1 response -file OUTPUT/out10
$conn1 screen -file OUTPUT/out11
$conn1 disconnect
```

Figure 8-1. Sample TermAsync Script 1

Let's assume there is an anonymous **ftp** archive that each week places new files in a directory called *new_this_week*. You can create a script (Figure 8-2) that each week accesses this archive and places them in a directory called */u/kjd/NEW*, assuming your home path is */u/kjd*.

```
xmyLoadPkg TermAsync
set conn2 [xmyTermAsync connect]
$conn2 wait -expect ": "
$conn2 sendWait "cd /users/kjb/NEW\r" -expect ": "
$conn2 sendWait "ftp ftp.bercco.com\r" -expect ": "
$conn2 sendWait "anonymous\r" -expect "d:"
$conn2 sendWait "kjb@\r" -expect "> "
$conn2 sendWait "cd pub/new_this_week\r" -expect "> "
$conn2 sendWait "prompt\r" -expect "> "
$conn2 sendWait "mget *\r" -expect "> "
$conn2 sendWait "bye\r" -expect ": "
$conn2 disconnect
xmyExit
```

Figure 8-2. Sample TermAsync Script 2

The script in Figure 8-3 logins to the NY Public Library and retrieves the title and author of the non-fiction best seller. The result is printed (using **xmyPrint**) in the output file.

```
# only user and error output
set xmyVar(OutputLevel) {user error}
xmyLoadPkg TermAsync

set nylib [xmyTermAsync connect -timeout 30 -shell /bin/sh]
$nylib sendWait "PS1=\"\$ \"\r" -expect "\$ "
$nylib wait -expect "\$ "

# connect with ny public library
$nylib sendWait "telnet nyplgate.nypl.org\r" -expect ": "
$nylib sendWait "nypl\r" -expect ": "
# choose DB
$nylib sendWait "1\r" -expect "s the key labeled \"Return.\""
$nylib sendWait "\r" -expect "m"

# choose best seller list
$nylib sendWait "5\r" -expect "m"

# choose non fiction
$nylib sendWait "2\r" -expect ": \033\[0m\033\[1m"

# make sure we go the right screen
set rightScreen [$nylib compare -region {3 2 34 1} -expect {\
        "List: NYT Nonfiction Best Sellers " }]

# if the screen is the one we want print title & author of
# best seller into log file

if  { $rightScreen } {
 set NF_BestSellerAuthor [$nylib screen -region {5 9 60 1}]
 set NF_BestSellerTitle [$nylib screen -region {6 9 60 1}]
 xmyPrint -text "Best Selling Book (non fiction): \
                 $NF_BestSellerTitle \n \
                by $NF_BestSellerAuthor "
}

# now logoff && disconnect

$nylib sendWait "q\r" -expect "m"
$nylib sendWait "so\r" -expect "m"
$nylib sendWait "7\r" -expect ": "
$nylib sendWait "3\r" -expect "\$ "
$nylib disconnect
```

Figure 8-3. Sample TermAsync Script 2

9. Term3270 Extension Package

9.1 Overview

The Term3270 Extension Package provides the functions necessary for interactions with the 3270 device.

The user interface to the 3270 SE is through a graphical terminal emulation for interactive sessions and through the main GUI or Command Line User Interface (CLUI) when running automated test cases to the background execution environment. These extensions will let you directly manipulate scripts to tailor the test case against the SUT.

> **NOTE —** To access the Term3270 Extension Package you must first run **xmyLoadPkg Term3270**.

9.1.1 Methods Overview

Section 9.5.1 contains detailed descriptions of the Term3270 Method extensions. The extensions are listed in alphabetical order (within each category). Table 9-1 lists the methods, organizing them in the general functional categories. Table 9-1 also gives a brief description of each extension and the section where the detailed description can be found.

Table 9-1. Term3270 Method Extensions (Sheet 1 of 2)

Category	Method	Description	Section
Connection	**connect**	Establishes a connection to the 3270 host from the MYNAH Tcl script.	9.5.1.2, Page 9–16
	disconnect	Destroys a connection made to the host through the **connect** method.	9.5.1.4, Page 9–19
Data Entry/Retrieval	**fieldLength**	Returns the length of the specified field.	9.5.1.7, Page 9–21
	screen	Returns the portion of the screen specified through location and dimensions.	9.5.1.16, Page 9–30
	send	Simulates the pressing of a 3270 function key.	9.5.1.17, Page 9–31
	type	Sends the keystrokes specified by a “text” parameter.	9.5.1.19, Page 9–33

Table 9-1. Term3270 Method Extensions (Sheet 2 of 2)

Category	Method	Description	Section
Location	**fieldBegin**	Moves the cursor to the first position of the field located by the given location parameters.	9.5.1.6, Page 9–20
	fieldNext	Moves the cursor to the first position of the next field.	9.5.1.8, Page 9–22
	find	Finds the specified string on the current screen, returning the row and column position of its location.	9.5.1.9, Page 9–23
	findLabel	Finds the specified label on the current screen, returning the row and column position of its location.	9.5.1.10, Page 9–24
	format	Loads tag name files used for finding a screen location through tagnames.	9.5.1.11, Page 9–25
	moveCursor	Moves the cursor to the position specified by the given parameters	9.5.1.15, Page 9–29
Comparisons	**compare**	Compares a region in the current 3270 display screen of a connection with a compare pattern body.	9.5.1.1, Page 9–14
	disableMask	Disables a mask object.	9.5.1.3, Page 9–18
	enableMask	Enables an already created mask object for a particular connection instance.	9.5.1.5, Page 9–19
	ignore	Defines a region to ignore during subsequent **compare** statements.	9.5.1.13, Page 9–27
Waiting	**sendWait**	Sends the special function key to the 3270 SUT and waits for the SUT's reply.	9.5.1.18, Page 9–32
	wait	Notifies that incoming data is being sent from the host and that the script should wait until the host has fully completed its transmission.	9.5.1.20, Page 9–34
Attributes	**getAttribute**	Finds the attribute byte value of a location in the connection object's display.	9.5.1.12, Page 9–26
	listAttributeTypes	Lists all of the attribute descriptors.	9.5.1.14, Page 9–28

9.1.2 Attributes Overview

Section 9.5.2 contains detailed descriptions of the Term3270 Attribute extensions. The extensions are listed in alphabetical order. Table 9-2 lists the extensions, organizing them in general functional categories. Table 9-2 also gives a brief description of each extension and the section where the detailed description can be found.

Table 9-2. Term3270 Attribute Extensions (Sheet 1 of 3)

Category	Attribute	Description	Section
Connection	**-connections**	Returns a list of currently open 3270 connections.	9.5.2.4, Page 9–37
	-host	Returns the name of the host.	9.5.2.9, Page 9–39
	-model	Outputs the name of the model the **xmyTerm3270** instance is configured.	9.5.2.17, Page 9–41
	-name	Specifies a name for connection handles.	9.5.2.18, Page 9–42
	-port	Outputs the name of the port number.	9.5.2.19, Page 9–42
	-queryConnection	Returns the connection's state.	9.5.2.20, Page 9–43
	-status	Returns the connection's status.	9.5.2.24, Page 9–44
	-timeout	Specifies amount of time that the connection will wait to receive a solicited screen.	9.5.2.26, Page 9–45
	-TN3270E	Supports TN3270E protocol for connecting to a host that begins transmission in TN3270E	9.5.2.27, Page 9–45

Table 9-2. Term3270 Attribute Extensions (Sheet 2 of 3)

Category	Attribute	Description	Section
Data Entry/Retrieval	**-collectKeyCount**	Indicates whether or not a function key count should be kept.	9.5.2.2, Page 9–36
	-dataBytesReceived	Number of bytes received.	9.5.2.5, Page 9–37
	-formatName	Contains the format/screen name of a connection's current screen.	9.5.2.7, Page 9–38
	-keyCount	Displays the number of program keys pressed.	9.5.2.12, Page 9–40
	-lastKeyPressed	Returns the name of the last function key pressed.	9.5.2.13, Page 9–40
	-tagDir	Indicates the path(s) to look for tag name files.	9.5.2.25, Page 9–44
Location	**-column**	Returns the cursor's current column position.	9.5.2.1, Page 9–36
	-row	Returns the cursor's current row position.	9.5.2.21, Page 9–43
Comparisons	**-compareInvisibleFields**	Indicates whether invisible fields should be processed.	9.5.2.3, Page 9–37
	-failedCompares	Returns the number of failed compares.	9.5.2.6, Page 9–38
	-goodCompares	Returns the number of good compares performed.	9.5.2.8, Page 9–38
	-masks	Returns a list of enabled masks for a connection.	9.5.2.16, Page 9–41
	-screenIdFile	Indicates the file from which the screen identification information is to be taken.	9.5.2.22, Page 9–43
	-warningCompares	Number of warnings generated during compares.	9.5.2.28, Page 9–45

Table 9-2. Term3270 Attribute Extensions (Sheet 3 of 3)

Category	Attribute	Description	Section
Waiting	**-initialWait**	Indicates whether an initial read should be performed during logon.	9.5.2.10, Page 9–39
	-initialWaitExpect	Supplies the string expression that **initialWait** should wait for.	9.5.2.11, Page 9–39
	-lastResponseTime	Returns the time between last **send** and new **screen** ready on the connection.	9.5.2.14, Page 9–40
	-lastTransmitTime	Returns the last time a **send** was done.	9.5.2.15, Page 9–41
Attributes	**-showAttributes**	Write application's screen attribute bytes to the *SUTimage* file.	9.5.2.23, Page 9–44

9.2 Language Commands Conventions/Definitions

The following are some of the conventions and definitions for the arguments used by the 3270 methods:

-character ***position***	***position*** can be a value in the range of 1 to whatever the last position is on the 3270 presentation space, the last position being determined by the type of model. For instance, for model 2, the character position could have a value of 1 to 1920 (24 rows x 80 columns).
-label ***label***	***label*** is a string used to search against the current 3270 screen for the purposes of finding the field associated to the label. When the label is found, usually a **fieldNext** is implicitly called with the proper label arguments, such as ***-direction, iteration*** or ***occurrence***, to find the location of the desired field.
-tag ***tagname***	***tagname*** denotes the ability to specify the location of a screen display as defined in user tag name files.
-offset ***offset***	***offset*** specifies the number of positions from the actual location specified, through tagname or label processing, that the intended action should be performed. For label processing, the ***offset*** determines the final position from where the command should start looking for the user specified field position. Offsets will not wrap to the next line; instead an exception of illegal argument will be incurred.
-position ***position***	***position*** lets you specify the location within a screen display by row and column values. ***position*** is a list consisting of two elements, first the row, then the column value, for example **-position {10 55}**.
-dimension ***dimension***	***dimension*** denotes the ability to specify the dimensions of a block. ***dimension*** consists of a list of two items, first the width, then the height, for example **-dimension {5 8}**.
-direction ***iteration***	This is used during label processing. The way this parameter works is that, from the label location, the argument finds the number of fields away from the label, specified by ***iteration***, in the direction from the field, specified by ***-direction***, and uses that field for whatever function it is to perform. ***-direction*** can be ***right***, ***left***, ***up***, and ***down***. ***iteration*** is an integer, e.g., a valid entry would be ***-right 12***.

-occurrence *occurrence*	***occurrence*** is used during label processing. ***occurrence*** is an integer value that specifies the i'th occurrence of the label field on the screen that should be used to locate the desired field.
-searchFrom	**-searchFrom** is an optional flag to label processing statements. If it is used, search begins at the current cursor position and proceeds to search for the label in the 3270 screen until it finds it or reaches the last position on the screen.
-unprotected	**-unprotected** is an optional flag to label processing statements. If it is used, the search finds the next instance of the label within the 3270 screen, regardless of whether the label's field is protected or unprotected.
-protectedField	**-protectedField** is an optional attribute for the **fieldNext** method that tells **fieldNext** to find the next protected field.
-region *region*	***region*** defines a block or region on the screen, and the location of that block within the screen. ***region*** is a list of four items: row, column, width, and height values.
-expect *pattern*	***pattern*** is string you expect to find or receive from a SUT.
-mask *maskhandle*	**-mask *maskhandle*** is used to denote a pattern that should be ignored during a comparison. The ***maskhandle*** is created using **xmyMask** (Section 6.2.10).
-ignore *region*	**-ignore *region*** is used to denote a region of the screen that is to be ignored during a comparison.
-file *filename*	Used with the **screen** method (Section 9.5.1.16), **-file** returns an empty string and the response is saved to the specified ***filename***.
-append	Used with **-file**, **-append** appends the response to the specified ***filename***. If this attribute is not used and ***filename*** is an existing file, **response** will overwrite the file.
-string *string*	***string*** is a string used to search against the current 3270 screen.
ERROR: Error Message	This denotes the error message returned from a command for a particular error.

9.3 Term3270 Location Processing

The Term3270 Package supports three methods for referring to physical screen locations on a synchronous terminal.

- Row and column coordinates that refer to actual screen locations. This method is useful when the other two are not available or not desirable.
- Label names for screen locations where the label names are determined by a string ‘label’ that appears on the screen at run time.
- Tag names for screen locations where the tag names are defined by the user in a Tag Name File.

The following subsections describe each of these methods, explaining how to use the method as well as the strengths and weaknesses of each method.

To help illustrate how you use each method, we will use the Term3270 **fieldNext** extension. **fieldNext** moves the cursor to the first position of the next unprotected field. For a complete description of **fieldNext** and its syntax, please see Section 9.5.1.8.

> **NOTE —** The examples in the following sub-sections assume you created an Term3270 connection using the variable **conn1**.

Figure 9-1 contains an example of a 3270 screen We will be using this screen to illustrate each location method. We've added row and column markers to aid our discussions.

```
          1         2         3         4         5         6         7         8
  12345678901234567890123456789012345678901234567890123456789012345678901234567890
  --------------------------------------------------------------------------------
1:                                                                  06/04/96 08:58
2: EXMP01                  EXMP ENTITY SELECTION SCREEN                      N
3:
4:
5:
6:
7:
8:
9:                           ENTITY   :  _
10:
11:                          RECONNECT:  _
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23: Copyright (C)  1989, 96  BELLCORE,  ALL RIGHTS RESERVED.
24:                                                                          (VC)
```

Figure 9-1. Example 3270 Screen

9.3.1 Row/Column Processing

The first, and simplest, method of referring to a screen location is by row and column. For example, to move the cursor to the RECONNECT: entry field, which is at row 11 and column 41, you could enter

```
$conn1 fieldNext -position {9 41}
```

As we said, this is the simplest way of referring to a screen location, but it is also the least reliable. With each new release, text locations often change. For example, a new field may be added between the ENTITY and RECONNECT entry fields in Figure 9-1. In this case, the statements that reference the RECONNECT: entry field would have to be rewritten to reflect the new row and column locations. If you're testing multiple releases, you must have a script for each release. Each script in turn may reference this field several times. Rewriting all of the row/column statements may be increasingly tedious.

9.3.2 Label Processing

The second method of referring to a screen location is by label processing. In label processing the screen locations for fields are determined by a string ***label*** that appears on the screen. To find the ENTITY: entry field from Figure 9-1 using label processing, you could enter

```
$conn1 fieldNext -label "ENTITY:"
```

fieldNext would search for the first occurrence of the string ***ENTITY:*** on the screen.

The default behavior of label processing is for the search to begin at the top left corner of the 3270 screen, proceeding left to right, top to bottom until it reaches the first occurrence of the label as a protected field, or until it reaches the end of the screen, in which case, an exception is thrown.

The different options that can be specified for label processing can be used to modify the default behavior.

- ***-direction iteration*** is used to specify which direction, and how many fields away, the desired field is located from the label. ***-direction*** can be ***right***, ***left***, ***up***, and ***down***. ***iteration*** is an integer, e.g., a valid entry would be ***-right 12***. The default is **-right 1**.
- **-occurrence** is used to find the i'th occurrence of the label on the screen. The default occurrence is 1.
- **-unprotected** is used to find the label string on the screen, regardless of whether the label is a protected or unprotected field. This is mainly used for unformatted screens. The default is to find only labels that belong to protected fields.

The following is an example of a 3270 Tcl command that makes use of these label processing options. It finds the second occurrence of the label string "FISH" on the 3270 screen, moves down 2 fields, moves to the right 4 fields, and types "TROUT" at the beginning of the field found at that location.

```
$conn1 type -label "FISH" -occurrence 2 -down 2 \
      -right 4 -text "TROUT"
```

But what if the screen changes? Let's take a look at a few scenarios

- The label and field simply move. If there is only one occurrence of the label on the screen, then your script will still be able to find the correct location. If there are more than one occurrences of the label on the screen, you may have to change the **-occurrence** option.
- The label and field stay at the same location, but the name of the field changes. All this may require is a simple search and replace in the script using your favorite editor to reflect the new name. If the length of the label has changed, you may will also have to add or change the ***-direction*** option, but you may be able to do this by a search and replace, too.

- The label stays at the same location, but its name changes and the location of the field moves. This will probably require a great deal more work, especially if you're performing tests on multiple releases.

Like row/column processing, label processing is very easy to set up, but, if the screens change, it takes less work to rewrite your scripts. Scripts using label processing are also easier to read since the location entries are the exact string as it appears on the screen and not (sometimes) cryptic row/column coordinates.

> **NOTE —** A label is any string on the screen contained in a protected field. The MYNAH Script Builder has the ability to generate labels for the various Term3270 commands, if Label processing option is selected. It is easier to let the Script Builder generate the labels. This way, the generated scripts would get executed without any errors. If there are any errors in execution of the generated code, please contact MYNAH Support. You may choose a label manually and code it in the script, but there exists a risk of the label getting rejected in a few cases.

9.3.3 Tag Name Processing

The third method of referring to a screen location is by tag name processing, where scripting statements are written using user-defined labels called **tags** that reference locations on a screen in place of row and column integer values.These definition reside in files called **Tag Name** files. Multiple scripts can reference these files. If the format of the screen changes, only the definitions of the tags related to the screen need to be updated. For example, to move to the RECONNECT: entry field on Figure 9-1 using tag name processing, assuming there is a tag for this screen called ***recon***, you could type

```
$conn1 fieldNext -tag recon
```

The MYNAH script accesses the appropriate *Tag Name* files to determine the row and column coordinates associated with the tag names in the script. The *Tag Name* files are loaded using the format method, as in

```
$conn1 format MainMenu
```

where ***MainMenu*** is a *Tag Name* file.

NOTE — A user or the MYNAH Administrator must have set up Tag Name files prior to anyone trying to use them. The MYNAH Administrator will then save the Tag Name file in a special directory, called the **TagDirectory,** used to contain Tag Name files. The steps for creating *Tag Name* files are found in Section 7 of the *MYNAH System Administration Guide*.

Like label processing, scripts using tag names are easier to read since you're using text, quite often related to what is actually on the screen, to determine a screen location.

When a field's position changes, only the row and column values associated with the tag names in the *Tag Name* files have to be changed rather than every reference to the affected row and column or label values in the scripts.

Unlike row/column and label processing, which can be used immediately, tag name processing requires a great deal of preliminary setup. As we mentioned earlier, they must first be created and then the MYNAH Administrator must place them in the *TagDir*. If a position changes, you must inform the MYNAH Administrator to make the necessary changes to the appropriate *Tag Name* file. If you use row/column and label processing you can immediately change your scripts.

Tag name processing also uses the *Screen Identification* file, which makes it possible to generate format/screen names automatically.

The *Screen Identification* file is needed to identify the format/screen the user is on and to generate a format statement during an initial script capture session using the GUI's Script Builder.

The *Screen Identification* file contains regular expression patterns (as defined for the **regex(3X)** program) in conjunction with screen location. As a user navigates through an application, the *Screen Identification* file determines if these two conditions exist

- The format/screen name exists on the format/screen being displayed
- A regular expression can be defined to uniquely describe the format/screen name.

If these conditions determine that a string matching a pattern is found at a particular location on the current screen, the system identifies that portion of the screen that matches the pattern as the format for the screen. Once this format is determined, a statement is generated, and this statement is then used to load a *Tag Name* file of the same name.

9.4 Waiting for a Response

The Term3270 Package contains two methods, **sendWait** and **wait**, that suspend the script execution until the SUT has returned with the proper response. The arguments to these methods are **-timeout** and **-expect**, where

- **timeout** is the amount of time the **wait** should wait before timing out and issuing a timeout. If the timeout is set to 0, the SE returns control back to the script.
- **expect** is the literal string to search for when data is received from the SUT. If data is available, control is returned back to the script. If no data is available, an exception is thrown (fails), and the time-out handler is called.

When no **expect** data is supplied, the **wait** will complete when the first receive is made from the SUT. When no **timeout** value is supplied, the default **timeout** value is used. Otherwise, when a **timeout** and **expect** are supplied, the **wait** command will wait until the host has transmitted the exact string specified in the **-expect** argument, or until the supplied **timeout** time has been reached, at which point the timeout handler will be called.

> **NOTE —** If your script has multiple **waits**, it may be best to combine them into one **wait** method with an **expect** rather than have the script pause as it processes each **wait**.

9.5 xmyTerm3270 Class

xmyTerm3270 is the Tcl class command providing language extensions that are necessary for interactions with the 3270 device. Its methods provide the basis of operations on a 3270 connection that you will need to create sufficiently sophisticated test cases in your scripts.

9.5.1 Methods

The Term3270 package contains the following methods.

9.5.1.1 compare

Syntax

```
handle compare -dimension dimension \
      -expect {body of compare pattern}\
      ?-ignore region? ... ?-ignore region? \
      ?-mask maskhandle? ... ?-mask maskhandle? \
      ?-outputLabel label?

handle compare -region region \
      -expect {body of compare pattern} \
      ?-ignore region? ... ?-ignore region? \
      ?-mask maskhandle? ... ?-mask maskhandle? \
      ?-outputLabel label?

handle compare -tag tagname -dimension dimension \
      -expect {body of compare pattern} ?-offset offset? \
      ?-ignore region? ... ?-ignore region? \
      ?-mask maskhandle? ... ?-mask maskhandle? \
      ?-outputLabel label?

handle compare -label label -dimension dimension \
      -expect {body of compare pattern} ?-offset offset? \
      ?-direction iteration? ?-occurrence occurrence? \
      ?-searchFrom? ?-unprotected?
      ?-ignore region? ... ?-ignore region? \
      ?-mask maskhandle? ... ?-mask maskhandle? \
      ?-outputLabel label?
```

Return

0 for failure

1 for success

Description

The **compare** method compares a region in the current 3270 display screen of a connection with the **compare** pattern body. When the location is not specified, **compare** uses the current cursor position. If a failure occurs, **compare** increments the connection's **-failedCompares** attribute by 1. If a success occurs, it increments the connection's **-goodCompares** attribute by 1.

If you use tagname processing (the third form shown above) and you set the **-dimension** attribute's width to zero (0), **compare** enters the width for this tagname entered in the tagname file.

The **-warning** flag tells the **compare** statement to log a warning should the comparison fail, rather than log a compare failure.

-outputLabel is used to specify a label for this particular **compare** command that will be written into the output file. The label from the output file can be used to refer back to the **compare** command in the input script. This is for cross reference purposes only.

Example

```
> set result [$conn1 compare -region {10 20 1 1} \
      -expect {{A}}

> set t1 {“MAIN MENU”}
> $conn1 compare -tag “title” -expect $t1
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- System error has occurred
- Error obtaining region from specified region values

9.5.1.2 connect

Syntax

```
xmyTerm3270 connect \
           ?-host hostname? \
           ?-model model #? \
           ?-port portnumber? \
           ?-compareInvisibleFields boolean? \
           ?-TN3270E boolean? \
           ?-timeout seconds? \
           ?-showAttributes boolean? \
           ?-initialWait boolean? \
           ?-initialWaitExpect string expression?
           ?-name connection name?
           ?-collectKeyCount boolean?
           ?-tagDir tagname directory?
           ?-screenIdFile screenid file
```

Return

Handle to **xmyTerm3270** instance

Description

The **connect** method is used to establish a connection to the 3270 host from the MYNAH Tcl script. **connect** returns a handle to a connection back to the script. Defaults are provided for port (23, or Telnet) and model number (2) should they not be specified within the script. The attribute list provides the instance of **xmyTerm3270Conn** class with initial values that will impact the configuration of this singular connection. Attribute values that can be set and are not supplied will attain their values after the loading of the 3270 extension package, where the class attribute values will be set depending on what is available through configuration files.

connect takes has the following options

-host	The name of the host.
-model	The name of the model for the connection.
-port	The name of the port number.
-compareInvisibleFields	Indicates whether invisible fields should be processed.
-TN3270E	Supports TN3270E protocol for connecting to a host that begins transmission in TN3270E.
-timeout	The amount of time that the connection will wait to receive a solicited screen.
-showAttributes	Indicates wether to write an application's screen attribute bytes to the *SUTimage* file.

-initialWait	Indicates whether an initial read should be performed during logon.
-initialWaitExpect	The string expression that **initialWait** should wait for.
-name	A name for the connection handle.
-collectKeyCount	Indicates whether or not a function key count should be kept.
-tagDir	Indicates the path(s) to look for tag name files.
-screenIdFile	Indicates the name of the screen Identification file.

NOTE — All boolean based options are either TRUE or FALSE.

On a per connection basis, the attributes **-initialWait**, **-host**, **-model**, **-port**, **-name,** and **-TN3270E** can only be set through this initial connect call.

connect automatically creates a handle name, using the form **.xmyTerm3270_*n*+**. Each successive connection increments the count, i.e., the first handle would be .**xmyTerm3270_1**, and the second would be .**xmyTerm3270_2**. While you can use these relative handle names, using the **set** command, as in the following example, creates an absolute handle name.

Example

```
> set conn1 [xmyTerm3270 connect -host pyib1 -model 2 \
                  -port 23]
.xmyTerm3270_1
```

Exceptions

If an exception is raised, the connection is not made.

- System error occurred, can’t connect to host
- Connection can not be made, maximum number of EHLLAPI sessions reached
- Time-out
- Licensing problems.

9.5.1.3 disableMask

Syntax

handle disableMask *list of maskhandles*

Return

No result

Description

The **disableMask** method disables a mask object that has been activated for a particular connection instance through the **enablemask** method. After **disablemask**, any subsequent compare statements on the connection will no longer ignore the pattern defined in the mask object.

Example

```
> $conn1 disableMask $mask1 $mask2 $mask3
```

Exceptions

- Mask handle not enabled on connection

9.5.1.4 disconnect

Syntax

handle disconnect

Return

No result

Description

The **disconnect** method destroys a connection made to the host through the **connect** method. Once the disconnect call has been made, the connection is no longer valid and can not be used. The handle is deregistered from the Tcl name space and is no longer recognized by the interpreter.

Example

```
> $conn1 disconnect
```

Exceptions

System error has occurred

9.5.1.5 enableMask

Syntax

handle enableMask {*list of mask handles*}

Return

No result

Description

The **enableMask** method enables an already created mask object, using **xmyMask** (Section 6.2.10, page 6–19), for a particular connection instance. Any subsequent compare statements on the connection will ignore the pattern defined in the mask object.

Example

```
> $conn1 enableMask $mask1 $mask2 $mask3
```

Exceptions

Illegal mask handle(s) given to command

9.5.1.6 fieldBegin

Syntax

handle `fieldBegin`

handle `fieldBegin -position` *position*

handle `fieldBegin -tag` *tagname* `?-offset` *offset*`?`

Return

No return

Description

The **fieldBegin** method moves the cursor to the first position of the field located by the given location parameters. When no location parameters are supplied, **fieldBegin** uses the current cursor position. The location given could be any position within that field.

Example

This moves the cursor to the beginning of the field found at location row = 2 and column =5.

```
> $conn1 fieldBegin -position {2 5}
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- System error has occurred

9.5.1.7 fieldLength

Syntax

```
handle fieldLength

handle fieldLength -position position

handle fieldLength -tag tagname ?-offset offset?\
     ?-protectedField?

handle fieldLength -label label
     ?-offset offset? ?-direction iteration?\
     ?-occurrence occurrence? -searchFrom? ?-unprotected?
```

Return

Length of the specified field

Description

The **fieldLength** method returns the length of the current field (or the target field) from the given location parameters. When no location parameters are supplied, **fieldLength** uses the current cursor position.

Example

```
> $conn1 fieldLength -label “PASSWORD:” -occurrence 1 \
     -right 1
8
```

9.5.1.8 fieldNext

Syntax

```
handle fieldNext

handle fieldNext ?-protectedField?

handle fieldNext -position position

handle fieldNext -tag tagname ?-offset offset?

handle fieldNext -label label ?-protectedField?
     ?-offset offset? ?-direction iteration?\
     ?occurrence occurrence? ?-searchFrom? ?-unprotected?
```

Return

No return

Description

The **fieldNext** method moves the cursor to the first position of the unprotected field specified by the given location parameters. When no location parameters are supplied, **fieldNext** uses the current cursor position.

Example

```
> $conn1 fieldNext -label “USERID:” -occurrence 1 \
     -right 1
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- System error has occurred

9.5.1.9 find

Syntax

```
handle find -string string ?-occurrence occurrence? \
                        ?-searchFrom?
```

Return

Position of string

Description

The **find** method finds the specified string on the current screen and returns the row and column position of its location. The **-occurrence** parameter is used to find the i'th occurrence of the given string on the screen. **-searchFrom** tells **find** to use the current cursor position; otherwise, the top lefthand corner is used as the starting point.

The search does not wrap the screen, i.e., the search stops when the lower right position is reached.

Example

```
> $conn1 find -string "system" -occurrence 1
{8 17}
```

Exceptions

String not found

9.5.1.10 findLabel

Syntax

```
handle findLabel -label label ?-occurrence occurrence? \
     ?-searchFrom? ?-unprotected?
```

Return

Position of label

Description

The **findLabel** method finds the specified label on the current screen and returns the row and column position of its location. The **-occurrence** parameter is used to find the i'th occurrence of the label on the screen. **-searchFrom** tells **findLabel** to use the current cursor position; otherwise, the top lefthand corner is used as the starting point. **-unprotected** tells **findLabel** to consider unprotected instances of the label on the screen; otherwise, only protected labels are considered.

The search does not wrap the screen, i.e., the search stops when the lower right position is reached.

Example

```
> $conn1 findLabel -label "CENTER" -occurence 2
{20 30}
```

Exceptions

Label not found

9.5.1.11 format

Syntax

handle format *screenid*

Return

No result

Description

The **format** method is used solely for the purpose of finding screen locations through tagnames, defined in tag name files. Tag name files are named after the screen id of the screen they correspond to.

If tagnames are used, this statement should follow any screen update and precede any commands done to that screen. The ***screenid***, a unique screen identifier sent from the host, is either located at the top screen label or in the user's *Screen Identification* file.

If no ***screenid*** argument is supplied, **format** does nothing.

Example

This **format** method loads the tag name file *clear*.

```
> $conn1 format clear
```

This **moveCursor** command will use a tag from the *clear* tag name file, called ***clear1tag***.

```
$conn1 moveCursor -tag "clear1tag" -offset 0
```

Exceptions

- Illegal screen tagname table format
- Screen does not correspond to tag name file

9.5.1.12 getAttribute

Syntax

```
handle getAttribute

handle getAttribute -position position

handle getAttribute -tag tagname ?-offset offset?

handle getAttribute -label label ?-offset offset? \
     ?-direction iteration? ?-occurrence occurrence? \
     ?-searchFrom? ?-unprotected?
```

Return

List of attribute descriptors

Description

The **getAttribute** method is used to find the attribute byte value of a location in the connection object's display, specified by the parameters to **getAttribute**. When location is not specified, **getAttribute** returns attribute value for the current cursor position.

Example

```
> $conn1 getAttribute -position {10 20}
{UNPROTECTED INVISIBLE}
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- System error has occurred
- No attribute byte was found (unformatted presentation space)

9.5.1.13 ignore

Syntax

```
handle ignore -dimension dimension

handle ignore -region region

handle ignore -tag tagname -dimension dimension \
     ?-offset offset?

handle ignore -label label -dimension dimension \
     ?-offset offset? ?-direction iteration? \
     ?-occurrence occurrence? ?-searchFrom? \
     ?-unprotected?
```

Return

No result

Description

The **ignore** method defines a region to ignore on a 3270 display screen This ignore region is active on all subsequent statements that permit the **-ignore** region option. Once a program function key is pressed, the ignore is no longer active. When location is not specified, **ignore** uses the current cursor position.

Example

```
> $conn1 ignore -tag “Dates” -offset 0 -dimension {1 1}
```

Exceptions

- Error performing screen location processing
- Illegal screen position

9.5.1.14 listAttributeTypes

Syntax

```
xmyTerm3270 listAttributeTypes
```

```
handle listAttributeTypes
```

Return

List of attribute descriptors and descriptions for each listing.

Description

The **listAttributeTypes** method lists all of the attribute descriptors and provides descriptions for each of the descriptors.

Example

```
> xmyTerm3270 listAttributeTypes
```

```
UNPROTECTED      - unprotected field
MODIFIED         - modified field
UNMODIFIED       - unmodified field
PROTECTED        - protected field
HIGHLIGHTED      - highlighted field
INVISIBLE        - invisible field
NUMERIC          - numeric field
```

9.5.1.15 moveCursor

Syntax

```
handle moveCursor -position position

handle moveCursor -tag tagname ?-offset offset?

handle moveCursor -label label ?-offset offset? \
     ?-direction iteration?\
     ?-occurrence occurrence? ?-searchFrom? ?-unprotected?
```

Return

No return

Description

The **moveCursor** method moves the cursor to the position specified in the given parameters in the particular connection's 3270 display. If the provided screen location is not a modifiable field, **moveCursor** behaves similarly to **fieldNext** by finding the next rightward modifiable field relative to the supplied position.

Example

```
> $conn1 moveCursor -tag “Login” -offset 0
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- System error has occurred

9.5.1.16 screen

Syntax

```
handle screen -dimension dimension \
     ?-ignore region? ... ?-ignore region? \
     ?-mask maskhandle? ... ?-mask maskhandle? \
     ?-file filename? ?-append?

handle screen -region region \
     ?-ignore region? ... ?-ignore region? \
     ?-mask maskhandle? ... ?-mask maskhandle? \
     ?-file filename? ?-append?

handle screen -tag tagname -dimension dimension \
     ?-offset offset? \
     ?-ignore region? ... ?-ignore region? \
     ?-mask maskhandle? ... ?-mask maskhandle? \
     ?-file filename? ?-append?

handle screen -label label -dimension dimension \
     ?-offset offset? ?-direction iteration?\
     ?-occurrence occurrence? ?-searchFrom? \
     ?-unprotected? \
     ?-ignore region? ... ?-ignore region? \
     ?-mask maskhandle? ... ?-mask maskhandle? \
     ?-file filename? ?-append?
```

Return

Portion of the screen specified through location and dimensions

Description

The **screen** method returns a block of data defined by the width and height parameter, starting at the location specified by the given location parameters. When no location parameters are supplied, **screen** uses current cursor position.

Example

```
> $conn1 screen -region {1 1 14 1}
“ENTER USER ID-”
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- Illegal region specified
- System error has occurred

9.5.1.17 send

Syntax

handle send *3270key*

Return

No result

Description

The **send** method simulates the pressing of a 3270 function key to the host for this particular connection. It does not wait for a response from the host, and script execution is resumed upon transmission of the key input.

The 3270 keys that can be taken as an argument to this function are shown in Table 9-3.

Table 9-3. 3270 Function Keys

pf1-24	attention	leftTab	cursorUp
pa1-3	sysReq	rightTab	cursorDown
clear	cent	cursorLeft	delete
enter	not	cursorRight	backspace
reset	home	eraseEOF	

Example

```
> $conn1 send pf-1
```

Exceptions

- Error inputting key
- System error has occurred

9.5.1.18 sendWait

Syntax

```
handle sendWait -key 3270key ?-expect tcl_expression? \
     ?-timeout seconds?
```

Return

No result

Description

The **sendWait** method sends a special function key to the 3270 SUT and waits for the SUT's reply before resuming execution of the script. A Tcl regular expression can be passed to provide the wait mechanism with a condition that, when found, should return control back to the script. If no regular expression is provide, the first response back from the host returns control back to the script. The same keys are supported as with the **send** method, but **sendWait** is only meaningful when the key pressed is one that causes the 3270 SUT to respond with screen(s) of data. The timeout attribute specifies the number of seconds to wait before **sendWait** returns control back to the script.

The 3270 keys that can be taken as an argument to this function are shown in Table 9-3.

Example

```
> set waitstring “INPUT APPLICATION NAME AND PRESS ENTER”
> $conn1 sendWait -key pf-3 -expect $waitstring
```

Exceptions

Timeout reached, SUT has not returned, timeout handler called

9.5.1.19 type

Syntax

```
handle type -text text

handle type -position position -text text

handle type -tag tagname -text text ?-offset offset?

handle type -label label -text text \
     ?-offset offset? ?-direction iteration?\
     ?-occurrence occurrence? ?-searchFrom? \
     ?-unprotected?
```

Return

No result

Description

The **type** method sends the keystrokes defined within the ***text***, beginning at the location specified by the parameters to this statement. If the location specified is not a modifiable field, an error will be generated and the keyboard will likely become locked. When location is not specified, **type** uses the current cursor position. **type** moves the cursor to the location where the typing has ended. In the event that **type** fails, the cursor position remains unchanged.

Example

```
> $conn1 type -label “login” -text “tktee99”
```

Exceptions

- Error performing screen location processing
- Illegal screen position
- System error has occurred
- Error entering data (keyboard locked, system busy, protected field, etc.)

9.5.1.20 wait

Syntax

handle wait ?-expect *string*? ?-timeout *seconds*?

Return

No result

Description

The **wait** method blocks the script until a screen is received from the host or returns immediately if a screen has already been received from the host.

After each screen is sent to the host, the connection keeps track of the number of screens received from the host that have not been waited on yet.

When **wait** is used without an **-expect** option, it performs one of the following:

- Returns as soon as any screen is received
- Returns immediately if a screen has already been received that has not been waited on yet
- Aborts when the **timeout** expires.

When **wait** is used with an **-expect** option, it performs one of the following:

- Returns as soon as a screen is received that contains the expected string
- Returns immediately if the last screen received has not been waited on yet ***and*** contains the expected string
- Aborts when the **timeout** expires.

If a **timeout** value of zero is specified, the **wait**

- Returns immediately if the correct screen has been received (depending on the **-expect** option)
- Aborts immediately if the correct screen has not been received.

Using a **timeout** value of zero is rarely necessary.

In general, a **send** method followed by a **wait** method is identical in behavior to a **sendWait** method, with the following exception.

In situations where a screen is sent and no screens are expected, the **wait** method is not needed. If a **sendWait** method is used, though, it will attempt to identify the fact that no screen is expected and return immediately. However, if a **send** is followed by a **wait** in this situation, the **wait** will time out waiting for a screen to be received.

Example

Following are four functionally identical logoff procedures that assume that after the **PF-3** key is sent, two screens are received from the host.

```
#
proc logoff1 {conn} {
   $conn send pf-3
   $conn wait
   $conn wait
}

proc logoff2 {conn} {
   $conn send pf-3
   $conn wait -expect “PRESS ENTER”
}

proc logoff3 {conn} {
   $conn sendWait -key pf-3
   $conn wait
}

proc logoff4 {conn} {
   $conn sendWait -key pf-3 -expect “PRESS ENTER”
}
```

Exceptions

Timeout reached, SUT has not returned, timeout handler called.

9.5.2 Attributes

For an attribute that can be set, if no value is ever specified, the default is obtained first from the value of the attribute at a class level, which initially had obtained its values from the SE configuration class object. Attributes on a class level derive their values from configuration files initially, but thereafter can be set in a script. In **Stateless** and **ConnOnly** modes, class attributes that are changed in a script will be reset to the value stored in configuration prior to the execution of the next script. Class level attribute values will affect subsequent instantiations of the class, which will inherit these values. Instance level attributes can be used to set attributes on a per connection basis, overriding any values inherited from the class level. Some attributes, however, can only be set at the time of connection and can not be changed after the connect call.

When attributes are used without being given a value, either on a class or instance level, the current value of that attribute is returned as output. Still other attributes cannot be set at all, and are used merely to display stored information about a connection at a particular moment. These attributes can not take arguments. Typically, with few exceptions, these attributes have no meaning on a class level. (W/R) denotes an attribute that is readable and writable. (R) means the attribute is only readable.

9.5.2.1 -column (R)

Syntax

handle `-column`

Description

The **column** attribute outputs the cursor's current column position.

9.5.2.2 -collectKeyCount (W/R)

Syntax

`xmyTerm3270 -collectKeyCount ?`*boolean*`?`

Decription

The **collectKeyCount** attribute indicates whether or not a function key (entered through the **send** or **sendWait** methods) count should be kept.

9.5.2.3 -compareInvisibleFields (W/R)

Syntax

handle -compareInvisibleFields ?*boolean*?

xmyTerm3270 -compareInvisibleFields ?*boolean*?

Description

The **compareInvisibleFields** attribute indicates whether invisible fields should be processed by statements done to a particular connection.

FALSE - don't process
TRUE - process.

9.5.2.4 -connections (R)

Syntax

xmyTerm3270 -connections

Return

List of currently open 3270 connections by their Tcl name

Description

The **-connections** attribute lists the names of all the 3270 connections currently available.

Example

```
> xmyTerm3270 -connections
{.xmy327001 .xmy327002 .xmy327003 .xmy327004}
```

9.5.2.5 -dataBytesReceived (R)

Syntax

handle -dataBytesReceived

Description

The **dataBytesReceived** attribute returns the number of bytes received in the last message from the host.

9.5.2.6 -failedCompares (R)

Syntax

handle `-failedCompares`

Description

The **failedCompares** attribute returns the number of failed compares performed so far on the connection.

9.5.2.7 -formatName (R)

Syntax

handle `-formatName`

Description

The **formatName** attribute returns the format/screen name of a connection's current screen. Format/screen name is initially found through the *Screen Identification* file, which maps a screen's identifier to the proper tag name file.

9.5.2.8 -goodCompares (R)

Syntax

handle `-goodCompares`

Description

The **goodCompares** attribute returns the number of good compares performed so far on the connection.

9.5.2.9 -host (W/R)

Syntax

handle -host

xmyTerm3270 -host ?*hostname*?

Description

The **host** attribute is used to output the name of the host the **xmyTerm3270** instance is connected. **host** cannot be used to reset host information. This attribute can only be set at connection time, as an argument to the **connect** class method.

9.5.2.10 -initialWait (W/R)

Syntax

handle -initialWait

xmyTerm3270 -intitalWait ?*boolean*?

Description

The **initialWait** attribute indicates whether an initial read should be performed during logon.

FALSE- don't read
TRUE - read.

Default: TRUE.

9.5.2.11 -initialWaitExpect (W/R)

Syntax

handle -initialWaitExpect

xmyTerm3270 -initialWaitExpect ?*string expression*?

Description

The **initialWaitExpect** attribute supplies the string expression that **initialWait** should wait for. No expression means wait for the first screen to be sent from the host.

9.5.2.12 -keyCount (R)

Syntax

```
xmyTerm3270 -keyCount
```

Description

The **keyCount** attribute displays the number of program keys pressed for all 3270 connections.

9.5.2.13 -lastKeyPressed (R)

Syntax

handle `-lastKeyPressed`

Description

The **lastKeyPressed** attribute indicates the name of the last program function key or enter key pressed.

9.5.2.14 -lastResponseTime (R)

Syntax

handle `-lastResponseTime`

Description

The **lastResponseTime** attribute returns the time between last **send** and when the new **screen** ready on the connection. The information stored in this attribute will also be used when logging performance output, the response time between a **send**, and when the SUT has responded back accordingly.

9.5.2.15 -lastTransmitTime (R)

Syntax

handle -lastTransmitTime

Description

The **lastTransmitTime** attribute returns the last time a **send** was done on this connection.

9.5.2.16 -masks (R)

Syntax

handle -masks

Description

The **masks** attribute returns a list of all of the mask handles associated to a particular connection through the **enableMask** instance method.

Example

```
> $conn1 -masks
.xmyMask01 .xmyMask02 .xmyMask03
```

9.5.2.17 -model (W/R)

Syntax

handle -model

xmyTerm3270 -model ?*model number*?

Description

The **model** attribute is used to output the name of the model the **xmyTerm3270** instance is configured. **model** cannot be used to reset model information. This attribute is useful when determining row or column from a 3270 display character position value. **model** can only be set at connection time, as an argument to the **connect** class method.

9.5.2.18 -name (W/R)

Syntax

handle -name

xmyTerm3270 connect -name *?connection Name?*

Description

In the first form, the **name** attribute returns the connection name for a specified handle, e.g., **.xmyTerm3270_*x***.

In the second form, the name of the connection handle can be changed at the time of connection.

9.5.2.19 -port (W/R)

Syntax

handle -port

xmyTerm3270 -port ?*port number*?

Description

The **port** attribute outputs the name of the port number to which the **xmyTerm3270** instance is connected. **port** cannot be used to reset port number information. This attribute can only be set at connection time, as an argument to the **connect** class method.

9.5.2.20 -queryConnection (R)

Syntax

handle `-queryConnection`

Description

The **queryConnection** attribute returns the connections's state.

- BUSY - The connection is busy
- READY - The connection is ready
- DOWN - The connection is down
- LOCKED - The keboard is locked

9.5.2.21 -row (R)

Syntax

handle `-row`

Description

The **row** attribute outputs the cursor's current row position.

9.5.2.22 -screenIdFile (W/R)

Syntax

handle `-screenIdFile` *?file?*

`xmyTerm3270 -screenIdFile` *?file?*

Description

The **screenIdFile** attribute indicates the file from which the screen identification information is to be taken. It can be set only at connect time.

9.5.2.23 -showAttributes (W/R)

Syntax

handle -showAttributes ?*boolean*?

xmyTerm3270 -showAttributes ?*boolean*?

Description

The **showAttributes** attribute writes the application's screen attribute bytes to the *SUTimage* file.

- TRUE - write
- FALSE - don't write.

9.5.2.24 -status (R)

Syntax

handle -status

Description

The **status** attribute returns the connection's status, whether it is alive(up) or not (down).

0 - The connection is down

1 - The connection is up.

9.5.2.25 -tagDir (W/R)

Syntax

handle -tagDir ?*path*?

xmyTerm3270 -tagDir ?*path*?

Description

The **tagDir** attribute indicates the path(s) to look for tag name files.

9.5.2.26 -timeout (W/R)

Syntax

handle -timeout ?*seconds*?

xmyTerm3270 -timeout ?*seconds*?

Description

The **timeout** attribute returns or sets the amount of time that the connection will wait to receive a solicited screen before returning control back to the script.

9.5.2.27 -TN3270E (W/R)

Syntax

handle -TN3270E

xmyTerm3270 -TN3270E ?*boolean*?

Description

The **TN3270E** attribute supports the TN3270E protocol for connecting to a host that begins transmission in TN3270E. For a connection, **TN3270E** can only be set at connect time.

9.5.2.28 -warningCompares (W/R)

Syntax

handle -warningCompares

Description

The **warningCompares** attribute returns the number of warnings generated during compares so far on the connection.

10. General Application-to-Application Tcl Language Extensions

The MYNAH General Application-to-Application (AppApp) extension package provides functionality necessary for interaction with a SUT thru an application specific interface.

10.1 Overview

The application specific interface, called the Interface Collector, contains the necessary logic to communicate with the SUT. Connections can be opened to the SUT and messages sent and received to and from it. The MYNAH System uses TCP/IP to communicate with the Interface Collector. The Interface Collector may use any protocol which the SUT understands. Figure 10-1 illustrates this interaction.

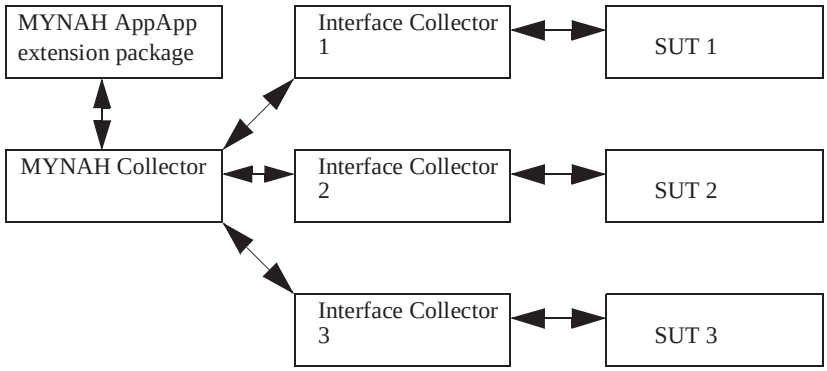

Figure 10-1. MYNAH General AppApp Interactions

The MYNAH Collector is a common MYNAH process used to manage a group of Interface Collectors. The MYNAH Collector can also manage other protocol handlers. These collectors are configured in the MYNAH configuration file, *xmyConfig*.

The MYNAH System supplies only a generic Interface Collector template. The application specific logic should be developed using the template.

> **NOTE —** To access the General AppApp extension package, you must first run **xmyLoad AppApp**.

The General AppApp extension package provides functionality to do the following:

- Make one or more logical connections to the SUT
- Send ASCII messages or files to the SUT
- Send binary files toSUT

- Wait for and receive ASCII messages from the SUT, saving them as files in an area called the Message Response Directory
- Receive binary message file from the SUT
- Filter unwanted incoming messages using user defined match procedures
- Open and scan all received files saved in the Message Response Directory. See the section on the Message Response Directory Tcl Language Extensions (Section 14.1).

Before using the General AppApp extension package, the MYNAH Collector and Interface Collector processes must be configured and running. Please refer to the *MYNAH System Administration Guide* for more information.

10.1.1 Methods Overview

Section 10.2.1 contains detailed descriptions of the AppApp Method extensions. The extensions are listed in alphabetical order (within each category). Table 10-1 lists the extensions, organizing them in general functional categories. Table 10-1 also gives a brief description of each extension and the section where the detailed description can be found.

Table 10-1. AppApp Method Extensions

Category	Method	Description	Section
Connection	**connect**	Establishes a logical connection to the MYNAH Collector and AppApp processes from the MYNAH Tcl script.	10.2.1.1, Page 10–5
	delete	Deletes a specified message or all messages for a connection	10.2.1.2, Page 10–7
	disconnect	Destroys a connection made through the **connect** method.	10.2.1.3, Page 10–8
Data Entry/Retrieval	**send**	Sends a message to the SUT using the AppApp connection established with the **connect** method.	10.2.1.5, Page 10–11
	receive	Returns a message from the SUT using the AppApp connection established using the **connect** method.	10.2.1.4, Page 10–9

10.1.2 Attributes Overview

Section 10.2.2 contains detailed descriptions of the AppApp Attribute extensions. The extensions are listed in alphabetical order (within each category). Table 10-2 lists the extensions, organizing them in general functional categories. Table 10-2 also gives a brief description of each extension and the section where the detailed description can be found.

Table 10-2. AppApp Attribute Extensions (Sheet 1 of 2)

Category	Attribute	Description	Section
Connection	**-connections**	Lists the names of all active connections.	10.2.2.3, Page 10–16
	-connId	Returns a unique identifier associated with the given connection	10.2.2.4, Page 10–17
	-IFhost	Returns the host on which the application specific Interface Collector is running.	10.2.2.7, Page 10–20
	-name	Lets you choose the name of the connection.	10.2.2.11, Page 10–26
Data Entry/Retrieval	**-append**	Instructs the **receive** operation to append a specified number of successfully received messages.	10.2.2.1, Page 10–13
	-data	Gets the message associated with the last **receive** method.	10.2.2.5, Page 10–18
	-file	Gets the name of the file containing the message associated with the last **receive** method.	10.2.2.6, Page 10–19
	-maxMsgs	Specifies the maximum number of messages that can be appended together by the **xmyMsgMatchUntil** procedure.	10.2.2.10, Page 10–25
Comparisons	**-listen**	Returns or sets the listen mode used when receiving messages.	10.2.2.8, Page 10–21
	-match	Specifies a Tcl procedure name that will be invoked for each incoming message processed by the **receive** method.	10.2.2.9, Page 10–23

Table 10-2. AppApp Attribute Extensions (Sheet 2 of 2)

Category	Attribute	Description	Section
Waiting	**-broadcast**	Permits all the waiting scripts to receive any messages received	10.2.2.2, Page 10–15
	-timeout	Sets the timeout for the **send** and **receive** operations.	10.2.2.18, Page 10–33
Attribute	**-recvPort**	Returns the port number.	10.2.2.12, Page 10–27
	-recvStatus	Returns the state of the receive session.	10.2.2.13, Page 10–28
	-recvTime	Returns the time stamp for the received message.	10.2.2.14, Page 10–29
	-sendPort	Returns the port number.	10.2.2.15, Page 10–30
	-sendStatus	Returns the state of the send session.	10.2.2.16, Page 10–31
	-sendTime	Returns the time the last message was successfully sent.	10.2.2.17, Page 10–32

10.2 xmyAppApp class

xmyAppApp is the Tcl class command providing language extensions that are necessary for automated interactions with the SUT using the General AppApp interface.

10.2.1 Methods

The AppApp package contains the following methods.

10.2.1.1 connect

Syntax

```
xmyAppApp connect -appName name ?-broadcast? \
     ?-listen listenOption? ?-match matchProc? \
     ?-maxMsgs maxMsgs? ?-name name? \
     ?-timeout timeout?
```

Returns

A handle name to the created **xmyAppApp** class instance.

Description

The **connect** method establishes a logical connection to the MYNAH collector and the local Interface Collector processes from the MYNAH Tcl script. Upon success, a handle to a connection is returned to the script. The attribute list provides the instance of **xmyAppApp** class with initial values that will impact the configuration of this connection. Attribute values not supplied with the connect method will obtain their values from the **xmyAppApp** command. If the value is undefined in the **xmyAppApp** class command or defined as the empty string, the corresponding value from the configuration file will be used.

connect takes the following attributes. These attributes are described in detail in subsequent sections.

-appName *name*	The name of the application, as defined in the *xmyConfig* configuration file. See the *MYNAH System Administration Guide* for details on the *xmyConfig* file.
-broadcast	Each received message is sent to all the waiting scripts, belonging to the same application.
-listen *listenOption*	Listen option for the **receive** method.
-match *matchProc*	The name of the Tcl procedure to be used for matching.
-maxMsgs *maxMsgs*	Maximum number of messages that can be appended.

-name *name*	Name of the application which the given Interface Collector is managing.
-timeout *timeout*	The time in seconds before script times out on receiving messages.

Example

```
> set A2A_conn1 [xmyAppApp connect -appName app_1]
```

Exceptions

Unable to establish the connection because

- **xmyCollector** process is not running or cannot be contacted
- Interface Collector process is not running or cannot be contacted
- **appName** is not defined in the *xmyConfig* file
- **appName** is not known to the **xmyCollector** process
- **appName** is not defined to use the AppApp protocol
- Timeout waiting for the connection back from the **xmyCollector** process

10.2.1.2 delete

Syntax

handle delete -file *fileName*

handle delete -file [*handle -file*]

handle delete -all

Returns

No result

Description

The **delete** method deletes the specified message or all messages for the connection identified by the **handle**.

delete takes the following attribute:

-file *fileName*	Deletes the message specified by fileName, which can be obtained by using the **-file** option on the **receive** method
-all	Deletes all messages for a connection. The **-all** option is usually used before disconnecting the connection.

Example

```
> $A2A_conn1 delete -filename 854028294.63.2
> $A2A_conn1 delete -all
```

10.2.1.3 disconnect

Syntax

handle `disconnect`

Returns

No result

Description

The **disconnect** method destroys the logical connection to the given Application made with the **xmyAppApp** connect class method and identified by the handle. Once the disconnect call is made, the handle name associated with the connection is no longer valid and will produce a Tcl “Invalid command name” error message if used.

Example

```
> $A2A_conn1 disconnect
```

10.2.1.4 receive

Syntax

```
handle receive ?-data? ?-file? ?-append? \
           ?-listen listenOption? ?-timeout timeout?
```

Returns

The received message if the **-data** attribute is specified, the filename containing the received message if the **-file** attribute was specified, otherwise no result.

Description

The receive method receives a message from the SUT using the AppApp connection established with the **connect** method and identified by *handle*. All messages received from the given Application from the Interface Collector will be saved in the Message Response Directory. Depending on the listen mode (see **-listen**), the receive operation will look for messages present in the Message Response Directory and/or wait for a message to arrive.

If a Tcl match procedure is defined (see **-match**), only messages that satisfy the match procedure will be returned by the receive operation.

Attributes

The attributes are described in detail in subsequent sections.

Example

In this example, the script waits a maximum of 300 seconds to receive a message.

```
> set message [$A2A_conn1 receive -data -timeout 300]
```

Side Effects

If the receive method was successful, the internal receive time (see **-recvTime**) and receive message variables (**-data** or **-file**) will be updated.

Exceptions

Unable to receive a message because:

- Timeout occurred while waiting for a message to arrive
- Invalid user defined Tcl match procedure
- No messages have been received, but the **-append** attribute was specified
- No messages have been received, but the MSG_LISTEN_NEXT mode was specified
- Unable to save received message to disk.

NOTE — The Message Response Directory must be writeable by the local Interface Collector processes. It is recommended that the person starting Interface Collector own both the Message Response Directory and the Interface Collector processes.

- Invalid or missing attribute values
- Both the **-data** and **-file** attributes were specified

10.2.1.5 send

Syntax

handle send -data *message -userData data*

handle send -file *filename -userData data*

Returns

No result

Description

The **send** method sends a message to the SUT using the AppApp connection established with the **connect** method and identified by ***handle***. The message is sent to the Interface Collector associated with the connection.

The message to be sent can be defined within the Tcl script or in a file. In the first case, the **-data** attribute must be used and the message is provided as the attribute value.

> **NOTE —** This message is treated like a string. If it contains backslash (`\`) or other special characters theymust be escaped with a backslash (‘\’).

In the second case, the **-file** option is used to specify the full path to a file containing the message.

> **NOTE —** Since this file name is passed to the Interface Collector for transmitting, the Interface processes must be able to open and read this file. When the Tcl script and Interface Collector are running on different machines, the file system containing the file to be sent must be accessible (mounted) by both the machines.

UserData can be used to send Application SUT specific information to the Interface Collector, which cannot be put in the message itself. For example, it could be priority of the message to be sent to be sent to Application SUT. The Interface Collector should interpret this information and act accordingly.

Example

In this example you send the string *sect{a=0;b=1;}% to the handle ***$A2A_conn1***

```
> $A2A_conn1 send -data “*sect{a=0;b=1;}%”
```

In this example you send the string *tag=\1\2\3%

```
> $A2A_conn1 send -data “*tag=\\1\\2\\3%”
```

In this example you send the file */mynah/scripts/AppApp/app_1.send.01*

```
> $A2A_conn1 send -file "/mynah/scripts/AppApp/app_1.send.01"
```

Side Effects

If the **send** method was successful, the internal send time variable (see **-sendTime**) will be updated.

Exceptions

Unable to send the message because

- Application send session is down. See **-sendStatus** attribute
- File is not accessible (if **-file** attribute was specified)
- Data message is too long, i.e. more than 2400 characters (if **-data** attribute was specified).

10.2.2 Attributes

The AppApp package contains the following attributes.

10.2.2.1 -append

Syntax

handle receive -append *number*

Returns

No result

Description

The **-append** attribute instructs the **receive** operation to append the next ***number*** of successfully received messages to the current received message before returning. The entire message will be accessible as the last received message.

The **-append** attribute is on a per receive basis and can only be specified with the **receive** method.

Example

This gets the first piece of a message sent in multiple pieces.

```
> $A2A_conn1 receive
```

Assume the first piece containsa number of subsequent pieces

```
regexp {(FRAGNUM=)([0-9]+)} [$A2A_conn1 -data] a b num
```

Receive and append the next pieces together with first piece

```
> $A2A_conn1 receive -append $num -listen MSG_LISTEN_NEXT
```

This is the entire message made of 1 + $num pieces

```
> set myMsg [$A2A_conn1 -data]
```

Example

This gets the first piece of a message sent in multiple pieces. Assume that the last piece contains the keyword LAST

```
> $A2A_conn1 receive
```

Receive and append the next pieces until the message contains the LAST keyword.

```
> while {[regexp {LAST} [$A2A_conn1 -data]] == 0} {
  $A2A_conn1 receive -append 1 -listen MSG_LISTEN_NEXT
}
```

This is the entire message.

```
> set myMsg [$A2A_conn1 -data]
```

Exceptions

Invalid **-append** attribute value.

10.2.2.2 -broadcast

Syntax

```
xmyAppApp connect -broadcast
handle -broadcast
```

Returns

The boolean value associated with the broadcast option.

Description

The **broadcast** option permits all the waiting scripts to receive any messages received by the Interface Collector, since the connection is made. This attribute can be set in the **connection** command only.

The received message may or may not be for the given connection. It is up to the script to filter out the message it receives.

Example

```
> $A2A_conn1 -broadcast
```

Exceptions

None

10.2.2.3 -connections

Syntax

```
xmyAppApp -connections
```

Returns

A blank separated list of active connection handle names (e.g., **.xmyAppApp_1**, **.xmyAppApp_2**, and **.xmyAppApp_4**), or the empty string if there are no active connections.

Description

The **-connections** attribute lists the names (**handle**) of all active (open) connections to AppApp. **-connections** can only be used through the **xmyAppApp** class command.

Example

```
> xmyAppApp -connections
.xmyAppApp_1 .xmyAppApp_2 .xmyAppApp_4
```

10.2.2.4 -connId

Syntax

handle `-connId`

Returns

The connection id associated with the given handle.

Description

The **-connId** attribute returns the connection id, which is is a unique identifier associated with the given connection. It is uniquely generated and maintained by the MYNAH collector process.

Example

```
> $A2A_conn1 -connId
```

Exceptions

None

10.2.2.5 -data

Syntax

handle -data

handle receive -data

Returns

The message associated with the most recent receive operation, or the empty string if no messages has been received yet.

Description

The **-data** attribute is used to get the message associated with the last **receive** method. If **-data** is used with the **receive** method, the received data will be returned. If the **receive** method fails, an exception will occur and the previously received message will remain unchanged.

Example

This checks if string “ORD=56700;” is in last received message.

```
> set msg [$A2A_conn1 -data]
> regexp {ORD=56700;} $msg
```

10.2.2.6 -file

Syntax

handle `-file`

handle `receive -file`

Returns

The filename containing the message associated with the most recent **receive** operation, or the empty string if no messages has been received yet.

Description

The **-file** attribute retrieves the name of the file, stored in the Message Response Directory, containing the most recent received message. The filename should be saved in the Tcl script in order to access this data if additional messages will be received. Since the Message Response Directory is purged on a regular basis, the contents of this file should be copied to a user area if the message need to be saved on a long term.

If **-file** is used with the **receive** method, the received filename will be returned. If the **receive** method fails, an exception will occur and the previously received filename will remain unchanged.

Example

Read the file containing the last received message.

```
> set fd [open [$A2A_conn1 -file] r]
> set data [read $fd]
> close $fd
```

10.2.2.7 -IFhost

Syntax

handle `-IFhost`

Returns

The host on which the application specific Interface Collector is running.

Description

The application specific Interface Collector runs on a specific host. The host name is defined in the *xmyConfig* file as “Host”. See the *MYNAH System Administration Guide* for details on the *xmyConfig* file.

Example

```
> $A2A_conn1 -IFhost
```

Exceptions

None

10.2.2.8 -listen

Syntax

```
xmyAppApp -listen ?listenMode?
xmyAppApp connect -listen listenMode
handle -listen ?listenMode?
handle receive -listen listenMode
```

Returns

When no value is specified it returns the current listen mode, otherwise no result.

Description

The **-listen** attribute returns or sets the listen mode, which determines what messages will considered by the **receive** method.

When no ***listenMode*** value is specified, **-listen** returns the current listen mode.

Only messages that have arrived **after** the listen time stamp will be considered by the **receive** operation. The valid **-listen** values are:

- **MSG_LISTEN_NOW** will set the listen time stamp to the current time. Thus, **receive** will return messages that have arrived after the time the **receive** operation was executed. All messages arrived prior to the receive operation will be ignored (but not removed from the Message Response Directory).
- **MSG_LISTEN_NEXT** will instruct **receive** to return the next message that arrived after the current received message. The listen time stamp will not be set. This mode is very useful for retrieving messages in sequence or messages that have arrived within the same second.
- **MSG_LISTEN_SEND** will set the listen time stamp to the time the most recent **send** operation was successfully executed. From a client point of view, we can assume that a reply (received message) to a request (sent message) cannot be received before the request itself is sent. On the other hand, if you are emulating the server side, then the receive must be performed first, so the **MSG_LISTEN_NOW** value should be used.
- An integer **time value** in UNIX format (i.e. seconds from 1/1/1970). The listen time stamp will be set to this given time, and messages that have arrived after this time will be considered.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is **MSG_LISTEN_NOW**.

-listen can be set in the *xmyConfig* file using the **ListenMode** parameter.

Example

Send a request and wait up to 30 seconds for the reply.

```
> $A2A_conn1 send -data $request_1

> set reply_1 [ $A2A_conn1 receive -timeout 30 -listen \
                 MSG_LISTEN_SEND -data]
```

Send the second request.

```
> $A2A_conn1 send -data $request_2

> set reply_2 [$A2A_conn1 receive -timeout 30 -listen \
                 MSG_LISTEN_SEND -data]
```

Example

In this example, three **sends** are done before performing any receives, so that a certain degree of parallelism is achieved between the client and server. The **MSG_LISTEN_SEND** wouldn't work properly because the first and second replies will be lost if they arrive before the third **send** operation is performed. The **MSG_LISTEN_NOW** will also not work if the replies arrive before the receive operation is executed.

Send three different requests expecting three replies.

```
> $A2A_conn1 send -data $request_1
> set saveTime [$A2A_conn1 -sendTime]   # save 1st send time
> $A2A_conn1 send -data $request_2
> $A2A_conn1 send -data $request_3
```

Get the replies.

```
> set reply_1 [$A2A_conn1 receive -data -listen $saveTime]
> set reply_2 [$A2A_conn1 receive -data \
                 -listen MSG_LISTEN_NEXT]
> set reply_3 [$A2A_conn1 receive -data \
                 -listen MSG_LISTEN_NEXT]
```

Exceptions

Invalid listen mode was specified

10.2.2.9 -match

Syntax

```
xmyAppApp -match ?matchTclProc?
xmyAppApp connect -match matchTclProc
handle -match
```

Returns

When no value is specified it returns the name of the Tcl procedure used by the receive operation to match (filter) all incoming messages, otherwise no result.

Description

The **-match** attribute specifies a Tcl procedure name that will be invoked for each incoming message processed by the **receive** method. Messages that do not satisfy the Tcl **match** procedure will not be returned by the receive operation. By default a **match** procedure is not defined and all messages can be returned by the **receive** method.

-match is set on a per connection basis and is set through the **xmyAppApp** class or by the **connect** method.

The user defined Tcl **match** procedure must take the connection handle name as its only argument and it returns a "1" when there is a match and "0" otherwise. Given the handle name, this match procedure can access the received data by using the ***-data*** attribute. Since this user defined match procedure is invoked for each message processed by the receive operation, it must be developed with performance considerations.

The existence and validity of the user defined Tcl match procedure will not be checked until the **receive** operation is executed.

> **NOTE —** Even though Tcl supports defining procedures named "(", "#", and other punctuation and special symbols, you should ***not*** define your Tcl **match** procedures with these names. Instead, meaningful procedure names should always be used.

-match can be set in the *xmyConfig* file using the **MatchProcedure** parameter.

Example

Receive only those messages containing “host=bluejays”.

```
> proc matchTclProc { $A2A_conn1 } {
      return [regexp {host=bluejays} [$A2A_conn1 -data]]
}
set conn1 [xmyAppApp connect -match matchTclProc ... ]
```

Exceptions

An attempt is made to change the match procedure using the instance handle

10.2.2.10 -maxMsgs

Syntax

```
xmyAppApp -maxMsgs ?number?
xmyAppApp connect -maxMsgs number
handle -maxMsgs ?number?
```

Returns

When no value is specified it returns the current message limit, otherwise no result.

Description

The **-maxMsgs** attribute specifies the maximum number of messages that can be appended together by the **xmyMsgMatchUntil** procedure. This parameter limits the number of messages appended before the until match conditions are satisfied.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is 10.

-maxMsgs can be set in the *xmyConfig* file using the **MaxMsgs** parameter.

Exceptions

An invalid **maxMsgs** value is specified

Example

```
> $A2A_conn1 -maxMsgs 25
```

10.2.2.11 -name

Syntax

`xmyAppApp connect -name` *`connectionName`*

`handle` `-name`

Returns

When no value is specified it returns the connection handle name set by the **connect** operation.

Description

The **-name** attribute lets you choose the name of the connection rather than having a name internally generated by the **connect** method.

The **-name** attribute is on a per connection basis and the value is set through the **connect** method.

Example

Create a connection named “my_connection”.

```
> xmyAppApp connect -name my_connection ...
```

Exceptions

- A connection with the same name already exists
- An attempt is made to change the connection name using the instance handle

10.2.2.12 -recvPort

Syntax

handle `-recvPort`

Returns

The port on which the application specific Interface Collector is communicating with the MYNAH collector process.

Description

The port on which the application specific Interface Collector is communicating with the MYNAH collector process. It is defined in the *xmyConfig* file as ***TcpPort***. See the *MYNAH System Administration Guide* for details on *xmyConfig* file.

The same port is used for both sending and receiving.

Example

```
> $A2A_conn1 -recvPort
```

Exceptions

None

10.2.2.13 -recvStatus

Syntax

handle -recvStatus

Returns

The value “UP” if the AppApp receive session is in TSOPEN state, otherwise returns “DOWN”.

Description

The **-recvStatus** attribute checks the state of the receive session between MYNAH and the SUT AppApp managers. “UP” is returned only if MYNAH’s AppApp manager is able to receive message from the SUT’s AppApp manager. Even though the receive status is “DOWN”, a **receive** may still be successful if the messages have already arrived andare stored in the Message Response Directory.

> **NOTE —** Defining and establishing (opening) the AppApp sessions are performed outside of the MYNAH System using AppApp commands.

Example

```
> $A2A_conn1 -recvStatus
```

10.2.2.14 -recvTime

Syntax

handle `-recvTime`

Returns

The time stamp in UNIX format (i.e., seconds since 1/1/1970) associated with the most recent received message, or 0 if no messages have been received.

Description

The **-recvTime** attribute returns the time stamp for the received message. All messages received from the SUT are stamped with the time the message was received by the MYNAH **collector** process.

Example

```
> $A2A_conn1 receive
> set rtime [$A2A_conn1 -recvTime]
```

10.2.2.15 -sendPort

Syntax

handle -sendPort

Returns

The port on which the application specific Interface Collector is communicating with the MYNAH collector process.

Description

The port on which the application specific Interface Collector is communicating with the MYNAH collector process. It is defined in the *xmyConfig* file as “TcpPort”.

The same port is used for both sending and receiving.

Example

```
> $A2A_conn1 -sendPort
```

Exceptions

None

10.2.2.16 -sendStatus

Syntax

handle -sendStatus

Returns

The value “UP” if the AppApp send session is in TSOPEN state, otherwise returns “DOWN”.

Description

The **-sendStatus** attribute returns the state of the send session between MYNAH and SUT AppApp managers. “UP” is returned only if the MYNAH System’s AppApp manager is able to **send** message to the SUT’s AppApp manager. If the **send** status is not “UP”, the **send** operation will fail.

> **NOTE —** Defining and establishing (opening) the AppApp sessions are performed outside of the MYNAH System using AppApp commands.

Example

```
> $A2A_conn1 -sendStatus
```

10.2.2.17 -sendTime

Syntax

```
handle -sendTime
```

Returns

The time stamp in UNIX format (i.e. seconds since 1/1/1970) associated to the most recent sent message, or 0 if no messages have been sent.

Description

The **-sendTime** attribute returns the time the last message was successfully sent over the connection.

Example

```
> $A2A_conn11 -sendTime
```

10.2.2.18 -timeout

Syntax

```
xmyAppApp -timeout ?timeoutValue?
xmyAppApp connect -timeout timeoutValue
handle -timeout ?timeoutValue?
handle send -timeout timeoutValue
handle receive -timeout timeoutValue
```

Returns

When no value is specified it returns the current timeout value, otherwise no result.

Description

The **-timeout** attribute sets the timeout for the **send** and **receive** operations. For the send operation, the timeout is the number of seconds to wait for acknowledgment of the sent message. If the timeout expires, the **send** operation will fail. For the **receive** operation, the timeout is the number of seconds to wait for a message to arrive. If the timeout expires, the receive operation will fail.

If a timeout value was not specified at connect time, the value in the configuration file for the particular protocol handler will be used. The timeout value can be changed for the connection instance, or changed for each individual send or receive operation.

-timeout can be set in the *xmyConfig* file using the **Timeout** parameter.

Example

Create a connection setting the timeout to five minutes.

```
> set A2A_conn1 [xmyAppApp connect -timeout 300 ]
```

The following **receive** waits at most five minutes

```
> $A2A_conn1 receive
```

The following **receive** waits at most ten minutes

```
> $A2A_conn1 -timeout 600
> $A2A_conn1 receive
```

The following **receive** waits at most one minute

```
> $A2A_conn1 receive -timeout 60
```

The following **receive** waits at most ten minutes

```
> $A2A_conn1 receive
```

10.3 Example

```
#  This script sends and receives messages to and from IF
#  collector.
xmyLoadPkg AppApp
xmyAppApp connect -appName app_1 -name a1 -timeout 10

set c1 [a1 -connId]
set listenValue MSG_LISTEN_SEND
set i 1

while {$i <= 500} {
# Send a message and receive. Since the test I/F collector
# is a loopback, the received message will be the same as the
# sent one.
   set msg “test message number $i to the I/F Collector”
   a1 send -data “$msg” -userData “Msg # $i”
   a1 receive -data -listen $listenValue
# Delete the first 100 messages.
   if {$i <= 100}  {
       a1 delete -file [a1 -file]
   }

   incr i
   set listenValue MSG_LISTEN_NEXT
}
# Delete the rest of the messages for the connection.
a1 delete -all

a1 disconnect
```

11. TOP Tcl Language Extension

The MYNAH TOP extension package provides functionality necessary for interactions with the SUT using the TOP/X.25 or TOP/TCP/IP protocols.

11.1 Overview

The TOP extension package interfaces with the data communications system called TOPCOM that handles the Transaction Oriented Protocols (TOP), as shown in Figure 11-1.

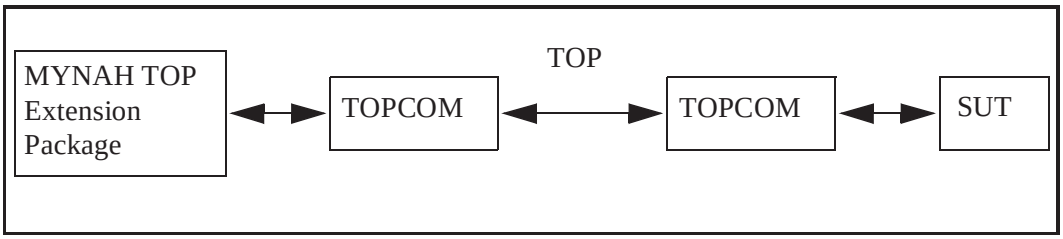

Figure 11-1. MYNAH TOPCOM Interactions

Users concerned with testing applications using TOP as the communications protocol may use the MYHAH TOP Tcl extension package and Tcl language to develop test scripts. The Tcl test scripts can be written to emulate an external application communicating with the application in the SUT.

> **NOTE —** To access the TOP Extension Package you must first run **xmyLoadPkg TOP**.

The TOP extension package provides functionality to

- Make one or more logical connections to the SUT over an established TOP session
- Send ASCII messages or files to the SUT
- Wait for and receive ASCII messages from the SUT, saving them as files in an area called the Message Response Directory
- Disconnect from a TOP session
- Filter unwanted incoming messages using user defined match procedures
- Perform TCIS or TCIS2 conversions on sent and received messages
- Analyze unformatted messages or messages in the Flexible Computer Interface Format (FCIF). See The section on the FCIF Tcl Language Extensions section.
- Open and scan all received files saved in the Message Response Directory. See the section on the Message Response Directory Tcl Language Extensions (Section 14.1).

Before using the TOP extension package, the TOPCOM and MYNAH Collector processes must be configured and running. Please refer to the *TOPCOM Guide for Administrators, Operators, & User* and the *MYNAH System Administration Guide* for more information.

The TOP Extension Package contains a process called a **Collector**, which stores all incoming messages into a special area (the **Message Response Directory**) on disk. The Collector is the interface between the MYNAH System and TOPCOM, sending and receiving messages from TOPCOM or any other protocol handler. A **handler** is the logical name of a TOPCOM entry defined in the *xmyConfig* file. All SEs performing a receive operation access the disk in order to get the message they are interested in. Whether or not you are concerned with checking received messages, the collector must be running.

The following is a list and description of the commands in the MYNAH TOP Extension Package.

11.1.1 Methods Overview

Section 11.2.1 contains detailed descriptions of the TOP Method extensions. The extensions are listed in alphabetical order (within each category). Table 11-1 lists the extensions, organizing them in general functional categories. Table 11-1 also gives a brief description of each extension and the section where the detailed description can be found.

Table 11-1. TOP Method Extensions

Category	Method	Description	Section
Connection	**connect**	Establishes a logical connection to the MYNAH Collector and TOPCOM processes from the MYNAH Tcl script.	11.2.1.1, Page 11–5
	disconnect	Destroys a connection made through the **connect** method.	11.2.1.2, Page 11–7
Data Entry/Retrieval	**send**	Sends a message to the SUT using the TOPCOM connection established with the **connect** method.	11.2.1.4, Page 11–10
	receive	Returns a message from the SUT using the TOPCOM connection established using the **connect** method.	11.2.1.3, Page 11–8

11.1.2 Attributes Overview

Section 11.2.2 contains detailed descriptions of the TOP Attribute extensions. The extensions are listed in alphabetical order (within each category). Table 11-2 lists the extensions, organizing them in general functional categories. Table 11-2 also gives a brief description of each extension and the section where the detailed description can be found.

Table 11-2. TOP Attribute Extensions (Sheet 1 of 2)

Category	Attribute	Description	Section
Connection	**-connections**	Lists the names of all active connections.	11.2.2.2, Page 11–14
	-dtn	Sets the TOPCOM Destination Transaction Name (DTN) parameter.	11.2.2.5, Page 11–17
	-name	Lets you choose the name of the connection.	11.2.2.11, Page 11–24
	-psn	Sets the TOPCOM Presentation Services Name (PSN) parameter.	11.2.2.12, Page 11–25
	-topcom	Specifies what TOPCOM handler to connect to.	11.2.2.20, Page 11–30
Data Entry/Retrieval	**-append**	Instructs the **receive** operation to append a specified number of successfully received messages.	11.2.2.1, Page 11–12
	-conversion	Tells the **send/receive** operation how to manipulate an application message.	11.2.2.3, Page 11–15
	-data	Gets the message associated with the last **receive** method.	11.2.2.4, Page 11–16
	-file	Gets the name of the file containing the message associated with the last **receive** method.	11.2.2.6, Page 11–18
	-maxMsgs	Specifies the maximum number of messages that can be appended together by the **xmyMsgMatchUntil** procedure.	11.2.2.9, Page 11–22
	-maxSegmentLen	Defines the maximum value of the two byte prefixes that is inserted at the beginning of each message segment during TCIS conversion.	11.2.2.10, Page 11–23

Table 11-2. TOP Attribute Extensions (Sheet 2 of 2)

Category	Attribute	Description	Section
Comparisons	**-listen**	Returns or sets the listen mode used when receiving messages.	11.2.2.7, Page 11–19
	-match	Specifies a Tcl procedure name that will be invoked for each incoming message processed by the **receive** method.	11.2.2.8, Page 11–21
Waiting	**-timeout**	Sets the timeout for the **send** and **receive** operations.	11.2.2.19, Page 11–29
Attribute	**-recvSession**	Returns the receive session number.	11.2.2.13, Page 11–26
	-recvStatus	Returns the state of the receive session.	11.2.2.14, Page 11–26
	-recvTime	Returns the time stamp for the received message.	11.2.2.15, Page 11–27
	-sendSession	Returns the send session number.	11.2.2.16, Page 11–27
	-sendStatus	Returns the state of the send session.	11.2.2.17, Page 11–28
	-sendTime	Returns the time the last message was successfully sent.	11.2.2.18, Page 11–28

11.2 xmyTop class

xmyTop is the Tcl class command providing language extensions that are necessary for automated interactions with the SUT using the TOP application-to-application interface.

11.2.1 Methods

The TOP package contains the following methods.

11.2.1.1 connect

Syntax

```
xmyTop connect -topcom “name” ?attribute list?
```

Returns

A handle name to the created **xmyTop** class instance.

Description

The **connect** method establishes a logical connection to the MYNAH Collector and the local TOPCOM processes from the MYNAH Tcl script. Upon success, a handle to a connection is returned to the script. The attribute list provides the **xmyTop** instance with initial values that will impact the configuration of this connection only. Attribute values not supplied with the **connect** method will obtain their values from the **xmyTop** class command. If the value is undefined in the **xmyTop** class command or defined as the empty string, the configuration file value will be used. All attribute values of **xmyTop** are configurable and can be changed by the user at connect time.

> **NOTE —** The ***name*** input to the **-topcom** attribute must match with a TOP protocol handler name defined in the *xmyConfig* file. (See the *MYNAH System Administration Guide* for information on the *xmyConfig* file.)

Attributes

The **-topcom** attribute is required with **connect**.

On a per connection basis, the following attributes can only be set in the **xmyTop** class or by the connect method: **-conversion**, **-match**, **-maxSegmentLen**, **-name**, **-topcom**. In other words, the values of these attributes can not be changed using an instance handle.

This is the complete list of connection class attributes that can be specified: **-conversion**, **-dtn**, **-listen**, **-match**, **-maxMsgs**, **-maxSegmentLen**, **-name**, **-psn**, **-timeout**, and **-topcom**.

Example

This example creates a connection to the TOPCOM handler **handler_1**.

```
> set TOP_conn [xmyTop connect -topcom “handler_1”]
```

In this example you try to connect to a TOPCOM handler that does not exist.

```
> set TOP_conn [xmyTop connect -topcom “foo”]
error: xmyTop connect: Protocol Handler (foo) not found in
configuration file
```

Exceptions

- Unable to establish the connection because
  - — **xmyCollector** process is not running or can not be contacted
  - — TOP handler name does not exist in configuration file
  - — TOP handler name is not known to the **xmyCollector** process
  - — TOP handler name is not defined to use the TOP protocol
  - — Timeout waiting for the connection back from the **xmyCollector** process.
- Invalid or missing attribute values

11.2.1.2 disconnect

Syntax

handle `disconnect`

Returns

No result

Description

The **disconnect** method destroys the logical connection to TOPCOM made with the **xmyTop connect** class method and identified by **handle**. Once the disconnect call has been made, the handle name associated with the connection is no longer valid and will produce a Tcl “Invalid command name” error message if used.

Example

```
> $TOP_conn disconnect
```

11.2.1.3 receive

Syntax

handle `receive ?-data? ?-file? ?attribute list?`

Returns

The received message if the **-data** attribute is specified, the filename containing the received message if the **-file** attribute is specified, otherwise no result.

Description

The **receive** command receives a message from the SUT using the TOPCOM connection established with the **connect** method and identified by **handle**. All messages received from TOPCOM (over the receive session number specified in the configuration file) will be saved in the Message Response Directory. Depending on the listen mode (see **-listen**), the receive operation will look for messages present in the Message Response Directory and/or wait for a message to arrive.

If a Tcl **match** procedure is defined (see **-match**), only messages that satisfy the **match** procedure will be returned by the **receive** operation.

Attributes

This is the complete list of **receive** attributes that can be specified: **-append**, **-data**, **-file**, **-listen**, **-timeout**. The values of these attributes will only impact the particular **receive** operation.

Example

In this example you wait a maximum of 300 seconds to receive a message.

```
> set message [$TOP_conn receive -data -timeout 300]
```

Side Effects

If the **receive** method was successful, the internal receive time (see **-recvTime**) and receive message variables (see **-data** or **-file**) will be updated.

Exceptions

- Unable to receive a message because
  - — timeout occurred while waiting for a message to arrive
  - — invalid user defined Tcl match procedure
- No messages have been received, but the **-append** attribute was specified
- No messages have been received, but the **MSG_LISTEN_NEXT** mode was specified
- Unable to save received message to disk

NOTE — The Message Response Directory must be writable by the local TOPCOM processes. It is recommended that the person starting TOPCOM own both the Message Response Directory and the TOPCOM processes.

- Invalid or missing attribute values
- Both the **-data** and **-file** attributes were specified

11.2.1.4 send

Syntax

handle send -data *message* ?attribute list?

handle send -file *filename* ?attribute list?

Returns

No result

Description

The **send** method sends a message to the SUT using the TOPCOM connection established with the **connect** method and identified by **handle**. The message is sent to the TOPCOM processes using the **send** session number defined in the configuration file.

The message to be sent can be defined within the Tcl script or in a file. In the first case, the **-data** attribute must be used and the message is provided as the attribute value.

> **NOTE —** This message is treated like a string. If it contains backslash (‘\’) or other special characters it must be escaped with a backslash (‘\’).

In the second case, the **-file** option is used to specify the full path to a file containing the message.

> **NOTE —** Since this file name is passed to TOPCOM for transmitting, the TOPCOM processes must be able to open and read this file. When the Tcl script and TOPCOM processes are running on different machines, the file system containing the file to be sent must be accessible (mounted) by both machines.

Attributes

This is the complete list of **send** attributes that can be specified: **-dtn**, **-psn**, **-timeout**. The values of these attributes will only impact the particular **send** operation.

Example

This example sends the string ****sect{a=0;b=1;agg{c=2;d=3;}}%*** to the handle **$TOP_conn**.

```
> $TOP_conn send -data “*sect{a=0;b=1;agg{c=2;d=3;}}%”
```

In this example you send the string ****tag=\1\2\3%***.

```
> $TOP_conn send -data “*tag=\\1\\2\\3%”
```

This time you send the file */mynah/scripts/Topcom/soac.send.01* to the handle **$TOP_conn**.

```
> $TOP_conn send -file "/mynah/scripts/Topcom/soac.send.01"
```

Side Effects

If the **send** method was successful, the internal send time variable (see **-sendTime**) will be updated.

Exceptions

- Unable to send the message because
  - — TOPCOM send session is down (not in TSOPEN state). See **-sendStatus** attribute
  - — Timeout occurred while waiting for a send acknowledgment from the SUT
- File is not accessible (if **-file** attribute was specified)
- Data message is too long, i.e., more than 200 characters (if **-data** attribute was specified).
- The **-psn** and **-dtn** attribute values are too long (if specified).

11.2.2 Attributes

The TOP package contains the following attributes.

11.2.2.1 -append

Syntax

handle receive -append *number*

Returns

No result

Description

The **-append** attribute instructs the **receive** operation to append the next ***number*** of successfully received messages to the current received message before returning. The entire message will be accessible as the last received message.

The **-append** attribute is on a per receive basis and can only be specified with the **receive** method.

Example

This gets the first piece of a message sent in multiple pieces.

```
> $TOP_conn receive
```

Assume the first piece contains a number of subsequent pieces

```
regexp {(FRAGNUM=)([0-9]+)} [$TOP_conn -data] a b num
```

Receive and append the next pieces together with first piece

```
> $TOP_conn receive -append $num -listen MSG_LISTEN_NEXT
```

This is the entire message made of 1 + $num pieces

```
> set myMsg [$TOP_conn -data]
```

Example

This gets the first piece of a message sent in multiple pieces. Assume that the last piece contains the keyword LAST

```
> $TOP_conn receive
```

Receive and append the next pieces until the message contains the LAST keyword.

```
> while {[regexp {LAST} [$TOP_conn -data]] == 0} {
  $TOP_conn receive -append 1 -listen MSG_LISTEN_NEXT
}
```

This is the entire message.

```
> set myMsg [$TOP_conn -data]
```

Exceptions

Invalid **-append** attribute value.

11.2.2.2 -connections

Syntax

```
xmyTop -connections
```

Returns

A blank separated list of active connection handle names (e.g., **.xmyTop_1**, **.xmyTop_2**, and **.xmyTop_4**), or the empty string if there are no active connections.

Description

The **-connections** attribute lists the names (**handle**) of all active (open) connections to TOPCOM. **-connections** can only be used through the **xmyTop** class command.

Example

```
> xmyTop -connections
.xmyTop_1 .xmyTop_2 .xmyTop_4
```

11.2.2.3 -conversion

Syntax

```
xmyTop -conversion ?conversionMode?
xmyTop connect -conversion conversionMode
handle -conversion
```

Returns

When no value is specified it returns the current conversion mode, otherwise no result.

Description

The **-conversion** attribute tells the **send/receive** operation how to manipulate the application message before *sending/returning* it to the SUT/script. **-conversion** is set on a per connection basis and is set throught the **xmyTop** class command or by the **connect** method.

The valid **-conversion** values are

- **MSG_TCIS** instructs the **send/receive** operation to automatically insert/remove the two bytes (non-ASCII) prefixes at/from the beginning of each segment of a message sent to/received from the SUT.
- **MSG_TCIS2** instruct the **send/receive** operation to automatically insert/remove the two bytes (non-ASCII) prefixes at/from the beginning of each segment of a message sent to/received from the SUT, where the message is composed of FCIF segments (Section 13).
- **MSG_EWNL** instructs the receive operation to append a newline character to each message received from the SUT. No conversion is done on the **send** operation.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is no conversion. The **-maxSegmentLen** attribute is used when converting a TCIS or TCIS2 message for sending.

-conversion can be set in the *xmyConfig* file using the **ConversionMode** parameter.

Example

```
> set TOP_conn [xmyTop connect -conversion MSG_TCIS ... ]
```

Example

```
> $TOP_conn -conversion
```

Exceptions

- An invalid conversion mode is specified
- An attempt is made to change the conversion mode using the instance handle

11.2.2.4 -data

Syntax

handle -data

handle receive -data

Returns

The message associated with the most recent receive operation, or the empty string if no messages has been received yet.

Description

The **-data** attribute is used to get the message associated with the last **receive** method. If **-data** is used with the **receive** method, the received data will be returned. If the **receive** method fails, an exception will occur and the previously received message will remain unchanged.

Example

This checks if string “ORD=56700;” is in last received message.

```
> set msg [$TOP_conn -data]
> regexp {ORD=56700;} $msg
```

11.2.2.5 -dtn

Syntax

```
xmyTop -dtn ?dtnValue?
xmyTop connect -dtn dtnValue
handle -dtn ?dtnValue?
handle send -dtn dtnValue
```

Returns

When no value is specified it returns the current DTN value, otherwise no result.

Description

The **-dtn** attribute defines the TOPCOM Destination Transaction Name (DTN) parameter in the TOP message sent to the SUT. The SUT's TOPCOM manager can use this value to determine the routing of the message.

-dtn can be set in the *xmyConfig* file using the **TopDefaultDTN** parameter.

Example

```
> $TOP_conn send -file "msg1.fcif" -dtn "APP1"
```

Exceptions

The DTN attribute value is too long.

11.2.2.6 -file

Syntax

handle `-file`

handle `receive -file`

Returns

The filename containing the message associated with the most recent **receive** operation, or the empty string if no messages has been received yet.

Description

The **-file** attribute retrieves the name of the file, stored in the Message Response Directory, containing the most recent received message. The filename should be saved in the Tcl script in order to access this data if additional messages will be received. Since the Message Response Directory is purged on a regular basis, the contents of this file should be copied to a user area if the message need to be saved on a long term.

If **-file** is used with the **receive** method, the received filename will be returned. If the **receive** method fails, an exception will occur and the previously received filename will remain unchanged.

Example

Read the file containing the last received message.

```
> set fd [open [$TOP_conn -file] r]
> set data [read $fd]
> close $fd
```

11.2.2.7 -listen

Syntax

```
xmyTop -listen ?listenMode?
xmyTop connect -listen listenMode
handle -listen ?listenMode?
handle receive -listen listenMode
```

Returns

When no value is specified it returns the current listen mode, otherwise no result.

Description

The **-listen** attribute returns or sets the listen mode, which determines what messages will considered by the **receive** method.

When no ***listenMode*** value is specified, **-listen** returns the current listen mode.

Only messages that have arrived **after** the listen time stamp will be considered by the **receive** operation. The valid **-listen** values are

- **MSG_LISTEN_NOW** will set the listen time stamp to the current time. Thus, **receive** will return messages that have arrived after the time the **receive** operation was executed. All messages arrived prior to the receive operation will be ignored (but not removed from the Message Response Directory).
- **MSG_LISTEN_NEXT** will instruct **receive** to return the next message that arrived after the current received message. The listen time stamp will not be set. This mode is very useful for retrieving messages in sequence or messages that have arrived within the same second.
- **MSG_LISTEN_SEND** will set the listen time stamp to the time the most recent **send** operation was successfully executed. From a client point of view, we can assume that a reply (received message) to a request (sent message) cannot be received before the request itself is sent. On the other hand, if you are emulating the server side, then the receive must be performed first, so the **MSG_LISTEN_NOW** value should be used.
- An integer **time value** in UNIX format (i.e. seconds from 1/1/1970). The listen time stamp will be set to this given time, and messages that have arrived after this time will be considered.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is **MSG_LISTEN_NOW**.

-listen can be set in the *xmyConfig* file using the **ListenMode** parameter.

Example

Send a request and wait up to 30 seconds for the reply.

```
> $TOP_conn send -data $request_1

> set reply_1 [ $TOP_conn receive -timeout 30 -listen \
                 MSG_LISTEN_SEND -data]
```

Send the second request.

```
> $TOP_conn send -data $request_2

> set reply_2 [$TOP_conn receive -timeout 30 -listen \
                 MSG_LISTEN_SEND -data]
```

Example

In this example, three **sends** are done before performing any receives, so that a certain degree of parallelism is achieved between the client and server. The **MSG_LISTEN_SEND** wouldn't work properly because the first and second replies will be lost if they arrive before the third **send** operation is performed. The **MSG_LISTEN_NOW** will also not work if the replies arrive before the receive operation is executed.

Send three different requests expecting three replies.

```
> $TOP_conn send -data $request_1
> set saveTime [$TOP_conn -sendTime]   # save 1st send time
> $TOP_conn send -data $request_2
> $TOP_conn send -data $request_3
```

Get the replies.

```
> set reply_1 [$TOP_conn receive -data -listen $saveTime]
> set reply_2 [$TOP_conn receive -data \
                 -listen MSG_LISTEN_NEXT]
> set reply_3 [$TOP_conn receive -data \
                 -listen MSG_LISTEN_NEXT]
```

Exceptions

Invalid listen mode was specified

11.2.2.8 -match

Syntax

```
xmyTop -match ?matchTclProc?
xmyTop connect -match matchTclProc
handle -match
```

Returns

When no value is specified it returns the name of the Tcl procedure used by the receive operation to match (filter) all incoming messages, otherwise no result.

Description

The **-match** attribute specifies a Tcl procedure name that will be invoked for each incoming message processed by the **receive** method. Messages that do not satisfy the Tcl **match** procedure will not be returned by the receive operation. By default a **match** procedure is not defined and all messages can be returned by the **receive** method.

-match is set on a per connection basis and is set through the **xmyTop** class or by the **connect** method.

The user defined Tcl **match** procedure must take the connection handle name as its only argument and it returns a “1” when there is a match and “0” otherwise. Given the handle name, this match procedure can access the received data by using the ***-data*** attribute. Since this user defined match procedure is invoked for each message processed by the receive operation, it must be developed with performance considerations.

The existence and validity of the user defined Tcl match procedure will not be checked until the **receive** operation is executed.

> **NOTE —** Even though Tcl supports defining procedures named “(”, “#”, and other punctuation and special symbols, you should ***not*** define your Tcl **match** procedures with these names. Instead, meaningful procedure names should always be used.

-match can be set in the *xmyConfig* file using the **MatchProcedure** parameter.

Example

Receive only those messages containing “host=bluejays”.

```
> proc matchTclProc { $TOP_conn } {
      return [regexp {host=bluejays} [$TOP_conn -data]]
}
set conn1 [xmyTop connect -match matchTclProc ... ]
```

Exceptions

An attempt is made to change the match procedure using the instance handle

11.2.2.9 -maxMsgs

Syntax

`xmyTop -maxMsgs` *`?number?`*

`xmyTop connect -maxMsgs` *`number`*

`handle` `-maxMsgs` *`?number?`*

Returns

When no value is specified it returns the current message limit, otherwise no result.

Description

The **-maxMsgs** attribute specifies the maximum number of messages that can be appended together by the **xmyMsgMatchUntil** procedure. This parameter limits the number of messages appended before the until match conditions are satisfied.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is 10.

-maxMsgs can be set in the *xmyConfig* file using the **MaxMsgs** parameter.

Exceptions

An invalid **maxMsgs** value is specified

Example

```
> $TOP_conn -maxMsgs 25
```

11.2.2.10 -maxSegmentLen

Syntax

```
xmyTop -maxSegmentLen ?number?
xmyTop connect -maxSegmentLen number
handle -maxSegmentLen
```

Returns

When no value is specified it returns the value of the current maximum segment length used during TCIS or TCIS2 conversion, otherwise no result.

Description

The **-maxSegmentLen** attribute defines the maximum value of the two byte prefix that is inserted at the beginning of each message segment. The two byte prefix contains the length of the segment it precedes, plus the length of the prefix (2 bytes). For **MSG_TCIS**, the message to be sent is divided into segments that are no more than **maxSegmentLen** - 2 bytes long. For **MSG_TCIS2** conversion, each FCIF section is considered a segment and will have a preceding two byte length prefix. FCIF sections greater than **maxSegmentLen** bytes long will be segmented If this attribute is not set by the Tcl script or configuration file, the default value is 65535 bytes.

The **-maxSegmentLen** attribute is set on a per connection basis and is set through the **xmyTop** class or by the **connect** method. The value of **-maxSegmentLen** is only used during the **send** operation when the conversion mode is set to **MSG_TCIS** or **MSG_TCIS2**.

-maxSegmentLen can be set in the *xmyConfig* file using the **MaxSegmentLen** parameter.

Example

```
> set conn1 [xmyTop connect -maxSegmentLen 65535 ... ]
```

Exceptions

An attempt is made to change the maximum segment length value using the instance handle.

11.2.2.11 -name

Syntax

`xmyTop connect -name` *`connectionName`*

`handle` `-name`

Returns

When no value is specified it returns the connection handle name set by the **connect** operation.

Description

The **-name** attribute lets you choose the name of the connection rather than having a name internally generated by the **connect** method.

The **-name** attribute is on a per connection basis and the value is set through the **connect** method.

Example

Create a connection named “my_connection”.

```
> xmyTop connect -name my_connection ...
```

Exceptions

- A connection with the same name already exists
- An attempt is made to change the connection name using the instance handle

11.2.2.12 -psn

Syntax

```
xmyTop -psn ?psnValue?
xmyTop connect -psn psnValue
handle -psn ?psnValue?
handle send -psn psnValue
```

Returns

When no value is specified it returns the current PSN value, otherwise no result.

Description

The **-psn** attribute sets the TOPCOM Presentation Services Name (PSN) parameter in the message being sent to the SUT. The SUT's TOPCOM manager can use this value to determine the routing of the message.

-psn can be set in the *xmyConfig* file using the **TopDefaultPSN** parameter.

Example

```
> $TOP_conn send -psn "APP2"
```

Exceptions

The **-psn** attribute value is too long.

11.2.2.13 -recvSession

Syntax

handle -recvSession

Returns

The TOPCOM receive session number for the connection associated with the handle.

Description

The **-recvSession** attribute returns the receive session number associated with the handle's connection. The TOPCOM receive session number is defined in the configuration file for each protocol handler that connections can be made to.

-recvSession can be set in the *xmyConfig* file using the **TopRecvSession** parameter.

Example

```
> $TOP_conn -recvSession
```

11.2.2.14 -recvStatus

Syntax

handle -recvStatus

Returns

The value "UP" if the TOPCOM receive session is in TSOPEN state, otherwise returns "DOWN".

Description

The **-recvStatus** attribute checks the state of the receive session between MYNAH and the SUT TOPCOM managers. "UP" is returned only if MYNAH's TOPCOM manager is able to receive message from the SUT's TOPCOM manager. Even though the receive status is "DOWN", a **receive** may still be successful if the messages have already arrived and are stored in the Message Response Directory.

> **NOTE —** Defining and establishing (opening) the TOP sessions are performed outside of the MYNAH System using TOPCOM commands.

Example

```
> $TOP_conn -recvStatus
```

11.2.2.15 -recvTime

Syntax

handle -recvTime

Returns

The time stamp in UNIX format (i.e., seconds since 1/1/1970) associated with the most recent received message, or 0 if no messages have been received.

Description

The **-recvTime** attribute returns the time stamp for the received message. All messages received from the SUT are stamped with the time the message was received by the MYNAH **collector** process.

Example

```
> $TOP_conn receive
> set rtime [$TOP_conn -recvTime]
```

11.2.2.16 -sendSession

Syntax

handle -sendSession

Returns

The TOPCOM send session number for the connection associated with the handle.

Description

The **-sendSession** attribute returns the **send** session number associated with the handle's connection. The TOPCOM **send** session number is defined in the configuration file for each protocol handler that connections can be made to.

-sendSession can be set in the *xmyConfig* file using the **TopSendSession** parameter.

Example

```
> $TOP_conn -sendSession
```

11.2.2.17 -sendStatus

Syntax

```
handle -sendStatus
```

Returns

The value “UP” if the TOPCOM send session is in TSOPEN state, otherwise returns “DOWN”.

Description

The **-sendStatus** attribute returns the state of the send session between MYNAH and SUT TOPCOM managers. “UP” is returned only if MYNAH’s TOPCOM manager is able to **send** message to the SUT’s TOPCOM manager. If the **send** status is not “UP”, the **send** operation will fail.

> **NOTE —** Defining and establishing (opening) the TOP sessions are performed outside of the MYNAH System using TOPCOM commands.

Example

```
> $TOP_conn -sendStatus
```

11.2.2.18 -sendTime

Syntax

```
handle -sendTime
```

Returns

The time stamp in UNIX format (i.e. seconds since 1/1/1970) associated to the most recent sent message, or 0 if no messages have been sent.

Description

The **-sendTime** attribute returns the time the last message was successfully sent over the connection.

Example

```
> $TOP_conn1 -sendTime
```

11.2.2.19 -timeout

Syntax

```
xmyTop -timeout ?timeoutValue?
xmyTop connect -timeout timeoutValue
handle -timeout ?timeoutValue?
handle send -timeout timeoutValue
handle receive -timeout timeoutValue
```

Returns

When no value is specified it returns the current timeout value, otherwise no result.

Description

The **-timeout** attribute sets the timeout for the **send** and **receive** operations. For the send operation, the timeout is the number of seconds to wait for acknowledgment of the sent message. If the timeout expires, the **send** operation will fail. For the **receive** operation, the timeout is the number of seconds to wait for a message to arrive. If the timeout expires, the receive operation will fail.

If a timeout value was not specified at connect time, the value in the configuration file for the particular protocol handler will be used. The timeout value can be changed for the connection instance, or changed for each individual send or receive operation.

-timeout can be set in the *xmyConfig* file using the **Timeout** parameter.

Example

Create a connection setting the timeout to five minutes.

```
> set TOP_conn [xmyTop connect -timeout 300 ]
```

The following **receive** waits at most five minutes

```
> $TOP_conn receive
```

The following **receive** waits at most ten minutes

```
> $TOP_conn -timeout 600
> $TOP_conn receive
```

The following **receive** waits at most one minute

```
> $TOP_conn receive -timeout 60
```

The following **receive** waits at most ten minutes

```
> $TOP_conn receive
```

11.2.2.20 -topcom

Syntax

`xmyTop connect -topcom` *`topcomHandlerName`*

`handle` `-topcom`

Returns

When no value is specified it returns the name of the TOPCOM handler, defined in the configuration file, the connection has been established to, otherwise no result.

Description

The **-topcom** attribute tells what TOPCOM handler to connect to among those defined in the MYNAH configuration file. The **-topcom** attribute is on a per connection basis and is set through the **connect** method.

This is a required **connect** attribute and no default is provided.

Example

Connect to the TOPCOM handler “handler_1”.

```
> set handler [xmyTop connect -topcom “handler_1” ...]
```

Exceptions

The TOPCOM handler doesn’t exist (see the connect method)

11.3 Examples

The following subsections contain example of using the TOP Tcl extensions.

11.3.1 Example 1

```
set dir_to_snd_from “/u/mynah/systp/TESTS/fnc/addam4/AOSt08”
set my_f_to_send “sacr”
xmyLoadPkg TOP
xmyTop connect -topcom sndsoac6 -name s6
puts "sendSession = [s6 -sendSession]"
puts "sendStatus = [s6 -sendStatus]"
puts "recvSession = [s6 -recvSession]"
puts "recvStatus = [s6 -recvStatus]"
s6 send -file "$dir_to_snd_from/$my_f_to_send"
set resp [s6 receive -listen MSG_LISTEN_SEND \
     -data -timeout 10]
puts "response=$resp"
s6 disconnect
```

11.3.2 Example 2

```
# Define the FCIF request message to be sent

set soacReq1 “*sect{a=1;b=2;aggr{a=3;b=4;}}%”

# Define a Tcl procedure to match the response message
# associated with the request message. Assume that the
# response message contains the FCIF tag “ORD” whose value
# starts with the digit 3 or 4 followed by at least one digit

proc procMatchResp1 { $TOP_conn } {

  return [regexp {ORD=(3|4)[0-9]+} [$TOP_conn -data]]

}

# Connect to the SOAC application named “Soac1” setting the
# timeout to 5 minutes and conversion mode to TCIS

set soacConn [xmyTop connect -topcom Soac1 \
     -timeout 300 -conversion MSG_TCIS]

# Send the request message

$soacConn send -data $soacReq1

# Wait for the response message that will satisfy the desired
# match procedure

if {[xmyMsgMatch -1 $soacConn procMatchResp1] == 0} {
     < we have timed out and did not receive a msg>
xmyExit

}

# The response message has arrived and is ready to be analyzed

set SoacResp1 [$soacConn -data]

<analyze the received message>

# Close the connection and remove the handler

$soacConn disconnect
```

12. PRT3270 Tcl Language Extensions

12.1 Overview

The MYNAH PRT3270 extension package provides functionality necessary for receiving printer messages (print jobs). The PRT3270 extension package must interface with some type of printer emulation software running in the UNIX environment. For example, there exists IBM 3270 emulation software that emulates an IBM 3287 printer in the UNIX environment. When the emulation software receives an IBM print job, it pipes the print job to a special process that is part of the MYNAH PRT3270 extension package, as shown in. Figure 12-1.

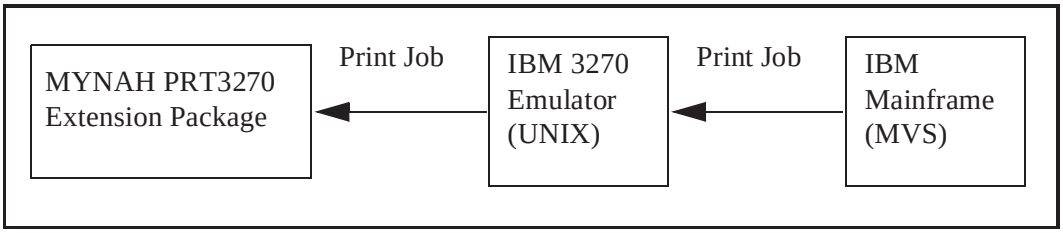

Figure 12-1. MYNAH PRT3270 Interactions

Users concerned with testing applications that send messages destined for printers may use the MYHAH PRT3270 Tcl extension package and Tcl language to develop test scripts. The Tcl test scripts can be written to mimic users looking for and verifying the contents of their print jobs.

> **NOTE —** To access the PRT3270 Extension Package you must first run **xmyLoadPkg PRT3270**.

The PRT3270 extension package provides functionality to:

- Make one or more logical connections to the printer emulator
- Wait for and receive print jobs from the SUT, saving them as files in an area called the Message Response Directory
- Disconnect from the printer emulator
- Filter unwanted print jobs using user defined match procedures
- Analyze the contents of the print job
- Open and scan all received print jobs saved in the Message Response Directory.

Before using the PRT3270 extension package, the printer emulation software (daemon process) must be configured and running. The IBM 3270 emulation products tested with MYNAH are:

- Sun Microsystems, SunLink SNA 3270 release 8.0
- Sun Microsystems, SunLink BSC 3270 release 8.0
- IO Concepts, 3287 Printer Client release 8.4.

The Prt3270 Extension Package contains a process called a **Collector**, which stores all incoming messages into a special area (the **Message Response Directory**) on disk. The Collector is the only process that receives messages from PrintCom (3270 Printer Emulator) or any other protocol **handler**, which is the logical name of of a PrintCom entry defined in the *xmyConfig* file. All SEs performing a receive operation access the disk in order to get the message they are interested in.

The following is a list and description of the commands in the MYNAH PRT3270 Extension Package.

12.1.1 Methods Overview

Section 12.2.1 contains detailed descriptions of the PRT3270 Method extensions. The extensions are listed in alphabetical order (within each category). Table 12-1 lists the extensions, organizing them in general functional categories. Table 12-1 also gives a brief description of each extension and the section where the detailed description can be found.

Table 12-1. PRT3270 Method Extensions

Category	Method	Description	Section
Connection	**connect**	Establishes a logical connection to the MYNAH Collector and 3270 Printer Emulator processes from the MYNAH Tcl script.	12.2.1.1, Page 12–5
	disconnect	Destroys a connection made through the **connect** method.	12.2.1.2, Page 12–7
Data Entry/Retrieval	**receive**	Returns a message from the SUT using the 3270 Printer Emulator connection established using the **connect** method.	12.2.1.3, Page 12–8

12.1.2 Attributes Overview

Section 12.2.2 contains detailed descriptions of the PRT3270 Attribute extensions. The extensions are listed in alphabetical order (within each category). Table 12-2 lists the extensions, organizing them in general functional categories. Table 12-2 also gives a brief description of each extension and the section where the detailed description can be found.

Table 12-2. PRT3270 Attribute Extensions (Sheet 1 of 2)

Category	Attribute	Description	Section
Connection	**-connections**	Lists the names of all active connections.	12.2.2.2, Page 12–12
	-name	Lets you choose the name of the connection.	12.2.2.9, Page 12–19
	-printcom	Specifies what PrintCom handler to connect to.	12.2.2.10, Page 12–20
Data Entry/Retrieval	**-append**	Instructs the **receive** operation to append a specified number of successfully received messages.	12.2.2.1, Page 12–10
	-conversion	Tells the **receive** operation how to manipulate an application message.	12.2.2.3, Page 12–13
	-data	Gets the message associated with the last **receive** method.	12.2.2.4, Page 12–14
	-file	Gets the name of the file containing the message associated with the last **receive** method.	12.2.2.5, Page 12–15
	-maxMsgs	Specifies the maximum number of messages that can be appended together by the **xmyMsgMatchUntil** procedure.	12.2.2.8, Page 12–18
Comparisons	**-listen**	Returns or sets the listen mode used when receiving messages.	12.2.2.6, Page 12–16
	-match	Specifies a Tcl procedure name that will be invoked for each incoming message processed by the **receive** method.	12.2.2.7, Page 12–17
Waiting	**-timeout**	Sets the timeout for the **receive** operation.	12.2.2.14, Page 12–23

Table 12-2. PRT3270 Attribute Extensions (Sheet 2 of 2)

Category	Attribute	Description	Section
Attributes	**-recvSession**	Returns the receive session number. This is always 0.	12.2.2.11, Page 12–21
	-recvStatus	Checks the state of the receive session. This is always ***up***.	12.2.2.12, Page 12–21
	-recvTime	Returns the time stamp for the last received message.	12.2.2.13, Page 12–22

12.2 xmyPrt3270 class

The methods for the **xmyPrt3270** class provides emulation of a 3270 printer so that you can capture messages that a SUT sends to a printer.

12.2.1 Methods

The PRT3270 package contains the following methods.

12.2.1.1 connect

Syntax

```
xmyPrt3270 connect -printcom name ?attribute list?
```

Returns

A handle name to the created **xmyPrt3270** class instance.

Description

The **connect** method establishes a logical connection to the MYNAH Collector and 3270 Printer Emulator processes from the MYNAH Tcl script. Upon success, a handle to a connection is returned to the script. The attribute list provides the **xmyPrt3270** instance with initial values that will impact the configuration of this connection only.

Attribute values that are not supplied with **connect** will obtain their values from the **xmyPrt3270** class. If the value is undefined in the **xmyPrt3270** class, or defined as the empty string, the configuration file value will be used. All attribute values of the **xmyPrt3270** class are configurable and can be changed by the user at connect time.

> **NOTE —** The ***name*** input to the **-printcom** attribute match with a PRT3270 protocol handler name defined in the *xmyConfig* file. (See the *MYNAH System Administration Guide*, for information on the *xmyConfig* file.)

Attributes

The **-printcom** attribute is required with **connect** .

On a per connection basis the following attributes can only be set in the **xmyPrt3270** class or by **connect** : **-conversion, -match, -name, -printcom**. In other words, the values of these attributes can not be changed using an instance handle (**handle**).

This is the complete list of connection class attributes that can be specified: **-conversion, -listen, -match, -maxMsgs, -name, -timeout**, and **-printcom**.

Example

This connects to the printer handler named ***printer-13***.

```
> set PRT_com [xmyPrt3270 connect -printcom “printer-13”]
```

Exceptions

- Unable to establish the connection because:
  - — xmyCollector process is not running or can not be contacted.
  - — PRT3270 handler name does not exist in configuration file.
  - — PRT3270 handler name is not known to the xmyCollector process.
  - — PRT3270 handler name is not defined to use the PRT3270 protocol.
  - — Timeout waiting for the connection back from the **xmyCollector** process.
- Invalid or missing attribute values.

12.2.1.2 disconnect

Syntax

handle disconnect

Returns

No result

Description

The **disconnect** method destroys the logical connection to the 3270 Printer Emulator made with the **connect** method and identified by **handle**. Once the disconnect call has been made, the handle name associated with the connection is no longer valid and will produce a Tcl “Invalid command name” error message if used.

Example

```
> $PRT_conn disconnect
```

12.2.1.3 receive

Syntax

handle receive *?attribute list?*

Returns

The received message (print job) if the **-data** attribute is specified, the filename containing the received message (print job) if the **-file** attribute is specified, otherwise no result.

Description

The **receive** method receives a message (print job) from the SUT using the 3270 printer emulator connection established with the **connect** method and identified by **handle**.

All messages received from the 3270 printer emulator will be saved in the Message Response Directory. Depending on the listen mode (see **-listen**, Section 12.2.2.6), the receive operation will look for messages present in the Message Response Directory and/or wait for a message to arrive.

If a Tcl match procedure is defined (see **-match**, Section 12.2.2.7), only messages that satisfy the match procedure will be returned by the receive operation.

Attributes

This is the complete list of receive attributes that can be specified: **-file, -timeout, -listen, -append, -data**. The values of these attributes will only impact the particular receive operation.

Example

This is an example of receiving a message with a timeout of 300 seconds

```
> set message [$PRT_conn receive -data -timeout 300]
```

Exceptions

- Unable to receive a message because:
  - — timeout occurred while waiting for a message to arrive.
  - — invalid user defined Tcl match procedure.
- No messages has been received, but the **-append** attribute was specified
- No messages has been received, but the **MSG_LISTEN_NEXT** mode was specified

- Unable to save received message to disk

  NOTE — The Message Response Directory must be writable by the printer emulation processes. It is recommended that the MYNAH Adminitrator own both the Message Response Directory and the printer emulation processes.

- Invalid or missing attribute values
- Both the **-data** and **-file** attributes were specified

12.2.2 Attributes

The PRT3270 package contains the following attributes.

12.2.2.1 -append

Syntax

handle receive -append number

Returns

No result

Description

The **-append** attribute instructs the receive operation to append the next ***number*** of successfully received messages to the current received message before returning. The entire message (print job) will be accessible as the last received message.

-append is set on a per receive basis and can only be specified with the **receive** method.

Example

This sets the first piece of a message sent in multiple pieces.

```
> $PRT_conn receive
```

This assumes the first piece contains number of subsequent pieces.

```
> regexp {(FRAGNUM=)([0-9]+)} [$PRT_conn -data] a b num
```

Receive and append the next pieces together with first piece,

```
> $PRT_conn receive -append $num -listen MSG_LISTEN_NEXT
```

This is the entire message made of 1 + $num pieces

```
> set myMsg [$PRT_conn -data]
```

Example

This gets the first piece of a message sent in multiple pieces. Assume that the last piece contains the keyword LAST.

```
> $PRT_conn receive
```

This receives and appends the next pieces until the message contains the LAST keyword.

```
> while {[regexp {LAST} [$PRT_conn -data]] == 0} {
  $PRT_conn receive -append 1 -listen MSG_LISTEN_NEXT
}
```

This is the entire message

```
> set myMsg [$PRT_conn -data]
```

Exceptions

Invalid **-append** attribute value.

12.2.2.2 -connections\

Syntax

```
xmyPrt3270 -connections
```

Returns

A blank separated list of active connection handle names (e.g., **.xmyPrt3270_1 .xmyPrt3270_2**, or **.xmyPrt3270_4**), or the empty string if there are no active connections.

Description

The **-connections** attribute lists the names (**handle**) of all active connections to a 3270 printer emulator. **-connections** can only be used through the **xmyPrt3270** class command.

Example

```
> xmyPrt3270 -connections
.xmyPrt3270_1 .xmyPrt3270_2 .xmyPrt3270_4
```

Example

```
> xmyPrt3270 -connections
```

12.2.2.3 -conversion

Syntax

```
xmyPrt3270 -conversion ?conversionMode?
xmyPrt3270 connect -conversion conversionMode
handle -conversion
```

Returns

When no value is specified it returns the current conversion mode, otherwise no result.

Description

The **-conversion** attribute tells the receive operation how to manipulate the application message before returning it to the script. The only mode available for the **xmyPrt3270** class is **MSG_EWNL**, which instructs the receive operation to append a newline character to each message (print job) received from the SUT.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is no conversion.

The **-conversion** attribute is set on a per connection basis and is set through the **xmyPrt3270** class or by the **connect** method.

-conversion can be set in the *xmyConfig* file using the **ConversionMode** parameter.

Examples

```
> set PRT_conn [xmyPrt3270 connect -conversion MSG_EWNL... ]
> $PRT_conn -conversion
```

Exceptions

- An invalid conversion mode is specified
- An attempt is made to change the conversion mode using the instance handle

12.2.2.4 -data

Syntax

handle -data

handle receive -data

Returns

The message (print job) associated with the most recent receive operation or the empty string if no messages have been received.

Description

The **-data** attribute returns the message associated with the most recent receive operation. If **-data** is used with the **receive** method, the received data will be returned. If **receive** fails, an exception occurs and the previously received message remains unchanged.

Example

This checks if the string “ORD=56700;” is in last received message,

```
> set msg [$PRT_conn -data]
> regexp {ORD=56700;} $msg
```

12.2.2.5 -file

Syntax

handle `-file`

handle `receive -file`

Returns

The filename containing the message (print job) associated with the most recent receive operation or the empty string if no messages has been received yet.

Description

The **-file** attribute returns the name of the file, stored in the Message Response Directory, containing the most recent received message. The filename should be saved in the Tcl script in order to access this data if additional messages will be received. Since the Message Response Directory is purged on a regular basis, the contents of this file should be copied to a user area if the message need to be saved on a long term.

If **-file** is used with the **receive** method, the received filename will be returned. If **receive** fails, an exception will occur and the previously received filename will remain unchanged.

Example

In this example you read the file containing the last received message.

```
> set fd [open [$PRT_conn -file] r]
> set data [read $fd]
> close $fd
```

12.2.2.6 -listen

Syntax

```
xmyPrt3270 -listen ?listenMode?
xmyPrt3270 connect -listen listenMode
handle -listen ?listenMode?
handle receive -listen listenMode
```

Returns

When no value is specified it returns the current listen mode, otherwise no result.

Description

The **-listen** attribute returns or sets the listen mode, which determines what messages (print jobs) will be considered by the **receive** method.

When no ***listenMode*** value is specified, **-listen** returns the current listen mode.

Only messages that have arrived ***after*** the listen time stamp are considered by the **receive** operation. The valid **-listen** values are

- **MSG_LISTEN_NOW** will set the listen time stamp to the current time. Thus, **receive** will return messages that have arrived after, the time the receive operation was executed. All messages arrived prior to the **receive** operation will be ignored (but not removed from the Message Response Directory).
- **MSG_LISTEN_NEXT** will instruct **receive** to return the next message that arrived after the current received message. The listen time stamp will not be set. This mode is very useful for retrieving messages in sequence or messages that have arrived within the same second.
- An integer **time value** in UNIX format (i.e. seconds from 1/1/1970). The listen time stamp will be set to this given time, and messages that have arrived after this time will be considered.

If this attribute is not set by the Tcl script or *xmyConfig* file, the default value is **MSG_LISTEN_NOW**.

-listen can be set in the *xmyConfig* file using the **ListenMode** parameter.

Example

```
> set job_1 [$PRT_conn receive -data -listen MSG_LISTEN_NOW ]
```

Exceptions

Invalid listen mode was specified

12.2.2.7 -match

Syntax

```
xmyPrt3270 -match ?matchTclProc?
xmyPrt3270 connect -match matchTclProc
handle -match
```

Returns

When no value is specified it returns the name of the Tcl procedure used by the **receive** operation to match (filter) all incoming messages, otherwise no result.

Description

The **-match** attribute specifies a Tcl procedure name that will be invoked for each incoming message processed by the **receive** method. Messages that do not satisfy the **match** procedure will not be returned by **receive**. By default a **match** procedure is not defined and all messages can be returned by **receive**.

The **-match** attribute is set on a per connection basis and is set through the **xmyPrt3270** class or by the **connect** method.

The user defined Tcl **match** procedure must take the connection handle name as its only argument and it must return a "1" when there is a match and "0" otherwise. Given the handle, this **match** procedure can access the received data by using the ***-data*** attribute. Since this user defined **match** procedure is invoked for each message processed by **receive**, it must be developed with performance considerations.

The existence and validity of the user defined Tcl **match** procedure will not be checked until the receive operation is executed.

> **NOTE —** Even though Tcl supports defining procedures named "(", "#", and other punctuation and special symbols, you should ***not*** define your Tcl **match** procedures with these names. Instead, meaningful procedure names should always be used.

-match can be set in the *xmyConfig* file using the **MatchProcedure** parameter.

Example

In this example you receive only those messages containing "host=bluejays".

```
> proc matchTclProc { $PRT_conn } {
  return [regexp {host=bluejays} [$PRT_conn -data]]
}
> set conn1 [xmyPrt3270 connect -match matchTclProc ... ]
```

Exceptions

An attempt is made to change the match procedure using the instance handle

12.2.2.8 -maxMsgs

Syntax

```
xmyPrt3270 -maxMsgs ?number?
xmyPrt3270 connect -maxMsgs number
handle -maxMsgs ?number?
```

Returns

When no value is specified it returns the current message limit, otherwise no result.

Description

The **-maxMsgs** attribute specifies the maximum number of messages that can be appended together inside the **xmyMsgMatchUntil** procedure. This parameter will limit the number of messages appended before the until match conditions are satisfied. The default value is 10.

-maxMsgs can be set in the *xmyConfig* file using the **MaxMsgs** parameter.

Exceptions

An invalid ***MaxMsgs*** value is specified

Example

```
> $PRT_conn -maxMsgs 25
```

12.2.2.9 -name

Syntax

`xmyPrt3270 connect -name` *`connectionName`*

`handle` `-name`

Returns

When no value is specified it returns the connection handle name set at connect time.

Description

The **-name** attribute lets you to choose the name of the connection rather than having a name internally generated by **connect** .

The **-name** attribute is set on a per connection basis and the value is set through the **connect** class.

Example

This is an example of creating a connection named **printer1**.

```
> xmyPrt3270 connect -name printer1...
```

Exceptions

- A connection with the same name already exists
- An attempt is made to change the connection name using the instance handle

12.2.2.10 -printcom

Syntax

```
xmyPrt3270 connect -printcom printcomHandlerName
handle -printcom
```

Returns

When no value is specified it returns the name of the 3270 printer emulation handler the connection has been established to, otherwise no result.

Description

The **-printcom** attribute tells what PRT3270 handler to connect to among those defined in the MYNAH configuration file. This is a required **connect** attribute and no default is provided.

The **-printcom** attribute is set on a per connection basis and is set through the **connect** method.

Example

This is an example of connecting to the print handler **handler_1**.

```
> set handler [xmyPrt3270 connect -printcom “handler_1” ...]
```

Exceptions

The print handler doesn’t exist (see the **connect** method)

12.2.2.11 -recvSession

Syntax

handle -recvSession

Returns

Always returns the value “0”.

Description

Unlike TOPCOM, session numbers are not used in the PRT3270 Extension Package. The **-recvSession** attribute was added to the **xmyPrt3270** class for consistency with the **xmyTop** class.

Example

```
> $PRT_conn -recvSession
```

12.2.2.12 -recvStatus

Syntax

handle -recvStatus

Returns

Always returns the value “UP”.

Description

If the MYNAH collector process is running, it will always be ready to receive messages from the printer emulation processes. Unlike TOPCOM, there are no sessions that can change status. The **-recvStatus** attribute was added to the **xmyPrt3270** class for consistency with the **xmyTop** class.

Example

```
> $PRT_conn -recvStatus
```

12.2.2.13 -recvTime

Syntax

handle -recvTime

Returns

The time stamp in UNIX format (i.e. seconds since 1/1/1970) associated with the most recent received message, or 0 if no messages (print jobs) have been received.

Description

The **-recvTime** attribute returns the time stamp for the received message. All messages (print jobs) received from the SUT are stamped with its arrival time by the MYNAH **xmyCollector** process.

Example

```
> $PRT_conn receive
> set rtime [$PRT_conn -recvTime]
```

12.2.2.14 -timeout

Syntax

```
xmyPrt3270 -timeout ?timeoutValue?
xmyPrt3270 connect -timeout timeoutValue
handle -timeout ?timeoutValue?
handle receive -timeout timeoutValue
```

Returns

When no value is specified it returns the current timeout value, otherwise no result.

Description

The **-timeout** attribute sets the timeout for the receive operation. For a **receive**, the timeout is the number of seconds to wait for a message to arrive. If the timeout expires, the **receive** operation fails.

If a timeout value was not specified at connect time, the value in the configuration file for the particular protocol handler will be used. The timeout value can be changed for the instance after the connection or changed for each individual receive operation.

-timeout can be set in the *xmyConfig* file using the **Timeout** parameter.

Example

Create a connection setting the receive timeout to five minutes.

```
> set handle [xmyPrt3270 connect -timeout 300 ... ]
```

The following **receive** waits at most five minutes.

```
> $PRT_conn receive
```

The following **receive** waits at most ten minutes

```
> $PRT_conn -timeout 600
> $PRT_conn receive
```

The following **receive** waits at most one minute

```
> $PRT_conn receive -timeout 60
```

The following **receive** waits at most ten minutes

```
> $PRT_conn receive
```

12.3 Example

Establish a connection for the printer named “printer_10”.

```
> set handle [xmyPrt3270 connect -printer “printer_10”]
```

Wait up to 10 minutes to receive a message.

```
> xmyMsgMatch 600 $PRT_conn procMatchMsg1
> set msg1 [$PRT_conn -data]
```

<analyze the message>

```
Close the connection and remove the handler.
```

> $PRT_conn disconnect

13. FCIF Tcl Language Extensions

13.1 Overview

This chapter describes the class command and methods available in the MYNAH System to process and analyze Flexible Computer Interface Format (FCIF) messages.

> **NOTE —** To access the FCIF Extension Package you must first run **xmyLoadPkg TOP**.

13.1.1 Methods Overview

Section 13.2.1 contains detailed descriptions of the FCIF Method extensions. The extensions are listed in alphabetical order (within each category). Table 13-1 lists the methods, organizing them in the general functional categories. Table 13-1 also gives a brief description of each extension and the section where the detailed description can be found.

Table 13-1. FCIF Method Extensions

Category	Method	Description	Section
Connection	**create**	Establishes a handle to an **xmyFcif** instance.	13.2.1.1, Page 13–3
	destroy	Removes a handle to an **xmyFcif** instance.	13.2.1.4, Page 13–10
Comparisons	**compare**	Compares the value of each tag in the master FCIF message to the corresponding value in the current FCIF message.	13.2.1.2, Page 13–5
	compareTags	Compares FCIF tags pulled out of the current message.	13.2.1.3, Page 13–7
	extraTags	Makes sure the current FCIF has no tags that are not present in the master FCIF.	13.2.1.5, Page 13–11
	getTag	Copies a substring of an FCIF value from the current FCIF.	13.2.1.6, Page 13–13
	reorder	Changes the order of like named sections or aggregates of both the current and master FCIF.	13.2.1.7, Page 13–14

13.2 xmyFcif Class

xmyFcif is the Tcl class command providing methods for manipulating and validating data that is in FCIF format. These methods support the following:

- Determining the differences between two FCIF strings. Output includes the presence of extra tags or missing tags.
- Creating a TVO (Tag Value Object) from an FCIF.
- Providing reorder of FCIF capability.

13.2.1 Methods

The FCIF package contains the following methods.

13.2.1.1 create

Syntax

```
xmyFcif create ?-file fileName? ?-fileMaster fileMasterName? \
               ?-data fcif? ?-dataMaster fcifMaster?
               ?-name handleName?
```

Returns

Handle to an **xmyFcif** instance.

Description

The **create** method creates a handle for processing FCIF messages. It takes either one or two FCIF messages and returns a handle that is then used with a method to perform the desired operation.

The FCIF messages can be provided by either specifying the filename containing the message or giving the message itself as a string. Any combination of filename and string arguments with the current and master messages is allowed.

At least one FCIF message is always required and is considered to be the current message; this is the message to be processed or analyzed. A second message is optional and is considered to be the master message. The master message acts as the base of comparison against the current message for certain methods, so it's not required unless one of these operations is performed.

> **NOTE —** The scope of **xmyFcif**'s handles is the script. Thus, handles cannot be passed to other scripts.

create takes the following options:

-file ***fileName***	Specifies the file containing the "current" message to be processed or analyzed. Either this option or **-data** ***fcif*** is required.
-fileMaster ***fileMasterName***	Specifies the file containing the "master" message that is to be compared against the current message that has been specified using either the **-file** or **-data** option.

-data *fcif*	Specifies the "current" message to be processed or analyzed. ***fcif*** can be an actual string or a variable set to the string. Either this option or **-file *fileName*** is required.
-name handleName	Specifies name of the handle. If not specified, a default name is generated.
-dataMaster *fcifMaster*	Specifies the "master" message that is to be compared against the current message that has been specified using either the **-file** or **-data** option. ***fcifMaster*** can be an actual string or a variable set to the string.

Example

In this example. the current message is in the file */tmp/MsgDir/file001*, and the master file is in */u/tom/file001.master*.

```
> set connFCIF [xmyFcif create -file /tmp/MsgDir/file001 \
                  -fileMaster /u/tom/file001.master]
```

In this example. the current message, **sect{a=3;b=2;c=4;}%*, has been assigned to a variable. Instead of comparing this message against another string, it is compared against a master string saved in the file */u/tom/file.master*.

```
> set message "*sect{a=3;b=2;c=4;}%"
> set connFCIF [xmyFcif create -data $message \
                  -fileMaster /u/tom/file.master]
```

Exceptions

- file(s) not found
- invalid FCIF message(s)

13.2.1.2 compare

Syntax

```
handle compare ?-excludedTags tagList? ?-warning?
```

Returns

The list of tags whose comparison failed

Description

This **compare** method verifies an FCIF message by comparing the value of each tag in the master FCIF message to the corresponding value in the current FCIF message. If a tag does not exist or the value does not match, this is considered a failed compare.

compare takes the following options:

-excludedTags *tagList*	Exclude some tags from the comparison. ***tagList*** is a list of tags that will not be searched and compared. For example, if the tester knows of a tag that varies between executions, such as a date, the tester can provide the tag name to the **compare** method using the **-excludedTags** options. The ***tagList*** format is defined below.
-warning	Increment **xmyVar(WarningCompares)**.

The ***tagList*** format is

```
{tag1 tag2 ... tagN}
```

where each tag has the following format

```
sectionName(occurence):aggregateName(occurence).tagName(occurence)
```

occurrence is the number of the section, aggregate, or tag occurrence. If omitted then the appropriate ***occurrence*** value will default to 1. If ***sectionName*** is omitted but its occurrence is specified then the *occurrence-th* unnamed section of the FCIF message will be taken into consideration.

Side Effects

compare increments the **goodcompare** and **failedcompare** counters. The warning counter will be incremented only if the **-warning** option was specified.

compare also produces “expected and actual data” in the script output compare file.

In the MYNAH version prior to 5.0 the functionality of this method was implemented by the directive **checktags**. This directive was affected by the **autoprt** flag. When the flag was ON **checktags** produced output into the log file for each failed compare and incremented the Error Count variable for each failed compare or incremented the Good Count variable if there was no failed compare.

Example

In this example we compare the current FCIF message against the master FCIF message. This example excludes from the comparison all tags containing a date value.We know that there are as many date tags as request aggregate plus a date tag for the FCIF message itself. These tags are in the first section only.

```
> set tagList [$connFCIF compare {sect:date \
                  sect:request(*).date}]
```

Exceptions

The master FCIF message was not specified at handle creation time.

13.2.1.3 compareTags

Syntax

```
handle compareTags -tag tagName -value value ?-offset int? \
          ?-length int? ?-operator operatorName?
```

Returns

A list of one or more element. The first element is the number of tags specified by the **-tag** option. Following elements, if any, are the name of the tags that failed the comparison.

Description

The **compareTags** method compares FCIF tags pulled out of the current message with the ***values*** provided by the user and returns the results of the comparison.

compareTags takes the following options:

-tag *tagName*	Specifies the FCIF tags to be checked. More than one tag can be checked by **compareTags** if more than one tag in the message satisfies the criteria given in the ***tagName*** argument. The ***tagName*** format is defined below.
-value *value*	Specifies the value provided to be compared with the FCIF tags.
-offset *int*	Defines the offset into the FCIF value. Used with **-length** to define the substring of the FCIF value. An offset of 0 is the beginning of the value. The default value is 0.
-length *int*	Defines the length of the FCIF value to be compared. Used with **-offset** to define the substring of the FCIF value. A length of 0 is the entire value. The default value is 0.

-operator ***operatorName*** Specifies the name of the relational operator to be used to compare the FCIF tag with the **-value** option. The following operator are defined:

- EQ (equal to)
- NE (not equal to)
- LT (less than)
- LE (less than or equal to)
- GT (greater than)
- GE (greater than or equal to)
- IN (substring of)
- NI (not substring of)

where the left operand is assumed to be the FCIF tag and the right operand is the value provided by the user. The default relational operator is EQ.

The **tagName** format is

```
sectionName(occurence):aggregateName(occurence).tagName(occurence)
```

occurrence is the number of the section, aggregate, or tag occurrence. If omitted then the appropriate ***occurrence*** value will default to 1. The ***occurrence*** can also be the character star (*). In such a case all existing ***occurrence*** will be taken into consideration. ***sectionName*** and its ***occurrence*** are optional and if omitted then all sections will be taken into consideration. If ***sectionName*** is omitted but its occurrence is specified then the *occurrence-th* unnamed section of the FCIF message will be taken into consideration.

Each of the **compareTags**' arguments can be a list of more than one value. This is used to perform more than one comparison with the same command. One of this application is the directive **fcifmatch** of the LMA language in the version of MYNAH prior to 5.0 (see the example).

Side Effects

- **compareTags** also produces "expected and actual data" in the script output compare file.
- Increments **xmyVar(GoodCompares)** and **xmyVar(FailedCompares**.

Example

```
> $connFCIF compareTags -tag sect:session.host -value bluejays
```

```
> $connFCIF compareTags -tag sect:session.host -length 8 \
                        -value bluejays

> $connFCIF compareTags -tag size -operator LT -value 512
```

This is an example of an implementation of the **fcifmatch** directive.

```
> $connFCIF compareTags -tag {C1 C1 C1 C1} \
                  -offset {1 7 8 19} \
                  -length {3 1 6 1} \
                  -value {PRE N 123R34 A}
```

This is an example of an implementation of the **sopr**e**spc** directive.

```
> $connFCIF compareTags -tag (*):SN -value $responseCode
```

13.2.1.4 destroy

Syntax

handle destroy

Returns

No result.

Description

The **destroy** method destroys a handle created by the **xmyFcif create** class method.

Example

```
> $connFCIF destroy
```

13.2.1.5 extraTags

Syntax

handle `extraTags ?-excludedTags tagList? ?-warning?`

Returns

The list of extra tags found in the current FCIF message.

Description

The **extratags** method verifies an FCIF message by making sure the current FCIF has no tags that are not present in the master FCIF.

extratags takes the following options:

-excludedTags *tagList*	Exclude some tags from analysis. ***tagList*** is a list of tags that will not be searched and compared. For example, if the tester knows of a tag that may appear but is not in the master FCIF, the tester can provide the tag name to the **compare** method using the **-excludedTags** options. The ***tagList*** format is defined below.
-warning	Increment **xmyVar(WarningCompares)**.

The ***tagList*** format is

```
{tag1 tag2 ... tagN}
```

where each tag has the following format

`sectionName(occurence):aggregateName(occurence).tagName(occurence)`

occurrence is the number of the section, aggregate, or tag occurrence. If omitted then the appropriate ***occurrence*** value will default to 1. The ***occurrence*** can also be the character star (*). In such a case all existing ***occurrence*** will be taken into consideration. ***sectionName*** and its ***occurrence*** are optional and if omitted then all sections will be taken into consideration. If ***sectionName*** is omitted but its occurrence is specified then the *occurrence-th* unnamed section of the FCIF message will be taken into consideration.

Side Effects

extratags increments **xmyVar(GoodCompares)** and **xmyVar(FailedCompares)**. The warning counter will be incremented only if the option ***-warning*** was specified.

Example

```
> set tagList [$connFCIF extratags {sect:agg(*).date \
                  sect:fragment}]
```

Exceptions

The master FCIF message was not specified at handle creation time.

13.2.1.6 getTag

Syntax

handle getTag -tag *tagName* ?-offset *int*? ?-length *int*?

Returns

The specified substring of the FCIF tag.

Description

The **getTag** method copies a substring of an FCIF value from the current FCIF and returns it as the return value of the method. A substring of the FCIF can be specified with the last two parameters.

getTag takes the following options:

-tag *tagName*	Specifies the substring with the name ***tagName***. The ***tagName*** format is defined below.
-offset *int*	Define the offset of a substring into the FCIF value. The default is 0.
-length *int*	Specifies the length of a substring. The default is 0.

The ***tagName*** has the following format:

```
sectionName(occurence):aggregateName(occurence).tagName(occurence)
```

occurrence is the number of the section, aggregate, or tag occurrence. If omitted then the appropriate ***occurrence*** value will default to 1. If ***sectionName*** is omitted but its occurrence is specified then the *occurrence-th* unnamed section of the FCIF message will be taken into consideration.

Example

This example gets the *host* tag in the session *aggregate* in the section *sect.*

```
> $connFCIF getTag -tag sect:session.host -length 8
```

This example gets the *C1* tag in the first unnamed section

```
> $connFCIF getTag -tag (1):C1
```

13.2.1.7 reorder

Syntax

handle reorder -aggregate *aggregateName ?*-key *tagList?*

Returns

No return.

Description

The **reorder** method changes the order of like named sections or aggregates of both the current and master FCIF, using values of tags inside the aggregate to determine the new ordering. This is useful when you have an application that sends like named aggregates in a random order. By using this directive, the tester can put the aggregate into a predictable order which makes the FCIF methods return accurate results.

reorder method takes the following options:

-aggregate *aggregateName*	Specifies the name of the section or aggregate that needs to be reordered. The ***aggregateName*** format is defined below.
-key *tagList*	Specifies a list of one or more tag names to be used as a key in reordering the aggregate. It can be any tag that is inside the aggregate to be reordered. The ***tagList*** format is defined below.

The ***aggregateName*** format is the following:

```
sectionName(occurence):aggregateName(occurence)
```

where the ***occurrence*** is the number of the section or aggregate occurrence. If omitted then it will be defaulted to 1. If the ***sectionName*** is omitted but its occurrence is specified then the ***occurrence-th*** unnamed section of the FCIF message will be taken.

The ***tagList*** format is the following:

```
aggregateName(occurence):tag(occurence)
```

More than one key tag can be used if a single tag is not sufficient to uniquely identify the ordering of the aggregate. This is needed when the value of the first key tag is the same in multiple aggregates.

Example

```
> $connFCIF reorder -aggregate sect(2).agg \
           -key {ordnum(1) ntu(1)}
```

14. Message Response Directory Tcl Language Extensions

The Message Response Directory extensions provide a mechanism to easily scan all messages that have arrived on a particular communications channel and have been saved to disk.

NOTE — For more information on Application to Application related extensions,

- See Section 10 for information on the General App-to-App extension package
- See Section 11 for information on the TOP extension package
- See Section 12 for information on the PRT3270 extension package
- See Section 13 for information on the Flexible Computer Interface Format (FCIF) extension package.

There are three categories of Message Response Directory Tcl Language Extensions. Table 14-1 lists these categories, including providing a description of each catergory.

Table 14-1. Message Response Directory Tcl Language Extension Categories

Section Title	Description	Section Number
xmyMsgDir class	Provides a mechanism to easily scan all messages that have arrived saved to disk.	14.1, Page 14–2
Match Tcl Extensions	Determine whether received messages match specified criteria.	14.2, Page 14–23
xmyMsgMarkFile	Mark/unmark a message in the Message Response Directory.	14.3, Page 14–28

NOTE — In incoming message, all null characters are changed to blanks.

14.1 xmyMsgDir class

The **xmyMsgDir** class provides a mechanism to easily scan all messages that have arrived on a particular communications channel and have been saved to disk into an area named the **Message Response Directory**. Since the space of the Message Response Directory is limited, a **Garbage Collector** needs to be developed by the user to remove the oldest messages when space is needed for new messages. Each message has an arrival time stamp associated to it. This time stamp, along with the time stamp (**time position**) set by the **xmyMsgDir** instance handle, is used to access the desired message. Users modify the time position of the handle in order to navigate into the Message Response Directory.

Attributes modifying the time position always take precedence over other attributes or commands specified on the same command line.

14.1.1 Methods

The Message Response Directory package contains the following methods.

14.1.1.1 close

Syntax

handleName `close`

Returns

No result

Description

The **close** method destroys the handle created through the **xmyMsgDir open** class method. Once the close call has been made, the handle name can no longer be used to scan the Message Response Directory, and will produce a Tcl “Invalid command name” error if used.

> **NOTE —** The message directory will be deleted if it is empty, i.e. if there are no more message.

Example

```
> $MSG_conn close
```

14.1.1.2 delete

Syntax

handle delete -file *fileName*

handle delete -all

Returns

No result

Description

The **delete** method deletes the specified message file within a given sub directory or delete all the messages within a given sub directory. The sub directory is specified in the **open** method.

delete takes the following attribute:

-file *fileName*	Deletes the message specified by fileName, which can be obtained by using the **-file** option on the **receive** method
-all	Deletes all messages for a connection. The **-all** option is usually used before disconnecting the connection.

Example

```
$MSG_conn delete -filename 854028294.63.2
$MSG_conn delete -all
```

NOTE — If the sub directory is not specified while opening the message directory and **-all** option is used for deleting, all the messages for the given application will be deleted. The **-all** option should be used with caution.

14.1.1.3 open

Syntax

```
xmyMsgDir open -topcom “handlerName” ?attribute list?
xmyMsgDir open -printcom “handlerName” ?attribute list?
xmyMsgDir open -handler “handleName” ?attribute list?
```

Returns

A handle (***handleName***) to the created **xmyMsgDir** class instance.

Description

The **open** method loads into memory the name of the files in the Message Response Directory received by the specified topcom or printcom protocol handler name. Each protocol handler name has a separate Message Response subDirectory for storing its received messages. Upon success, a handle is returned to the script.

The attributes that can be specified are **-marked**, **-subDir, -maxMsgs**, **-printcom**, and **-topcom**.

Remarks

Messages of all mark types (see **-marked**, Section 14.1.2.6) are loaded. Messages are loaded in order from newest to oldest.

Either the **-topcom** or **-printcom** or **-handler** attribute is required with the **open** method. The attribute value (handler name) is the same value used in the **xmyTop connect** and **xmyPrt3270 connect** methods and must match with a protocol handler entry in the configuration file.

On a per handle basis the following optional attribute can only be set through this initial open call: **-maxMsgs**.

The attributes thatcan be specified are **-marked**, **-maxMsgs**, **-printcom**, and **-topcom**.

Example

This creates a handle to scan messages received by the TOPCPM handler named **handler_1**.

```
> set MSG_conn [xmyMsgDir open -topcom handler_1]
```

Exceptions

- Unable to access the Message Response Directory
- Unable to open and load files from the Message Response Directory because handler name does not exist in the configuration file.

14.1.2 Attributes

The Message Response Directory package contains the following attributes.

14.1.2.1 -data

Syntax

handleName `-data`

Returns

The data associated with the message at the handle's current time position, or the empty string if the current position has not been set.

Description

The **-data** attribute is used to return the message at the current time position in the Message Response Directory. The **-first**, **-last**, and **-position** attributes can be used to initially set the handle's time position.

Example

This retrieves the first (newest) message in the Message Response Directory.

```
> $MSG_conn -first
> set message [$MSG_conn -data]
```

14.1.2.2 -file

Syntax

handleName `-file`

Returns

The file name containing the message at the handle's current time position or the empty string if the current position has not been set.

Description

The **-file** attribute returns the file name containing the message at the current time position in the Message Response Directory. The **-first**, **-last**, and **-position** attributes can be used to initially set the handle's time position.

Example

This gets the current Message Response Directory file name.

```
> set fileName [$MSG_conn -file]
```

14.1.2.3 -first

Syntax

handleName `-first`

Returns

No result

Description

The **-first** attribute sets the handle's current time position to the first (newest) message in the Message Response Directory based on the **-marked** attribute value.

Example

This positions the handle to the newest received message in the Message Response Directory.

```
> $MSG_conn -first
```

Exceptions

A first message does not exist because the message table is empty.

14.1.2.4 -handler

Syntax

```
xmyMsgDir open -handler generalPurposeHandlerName
```

Returns

No result.

Description

The **open** attribute is on a per handle basis and is set only through the **xmyMsgDir** open class method. Given the ***generalPurposeHandlerName***, a search is performed in the configuration file to determine the location (Message Response Directory) of the received messages to load. Since the General App-to-App protocol does not use session numbers, the default value is 0. The ***generalPurposeHandlerName*** is the same value that is required for the **xmyAppApp connect** class method

Example

```
# create a handle to get messages from the printcom handler
# named “printer_1”

> set MSG_conn [xmyMsgDir open -handler app_1]
```

Exceptions

The specified handler does not exist in the configuration file.

14.1.2.5 -last

Syntax

handleName `-last`

Returns

No result

Description

The **-last** attribute sets the handle's current time position to the last (oldest) loaded message in the Message Response Directory based on the **-marked** attribute value

Example

This gets the file name containing the oldest message received.

```
> $MSG_conn -last
> set fileName [$MSG_conn -file]
```

Exceptions

A last message does not exist because the message table is empty.

14.1.2.6 -marked

Syntax

xmyMsgDir open -marked markType

handleName -marked ?*markType?*

Returns

When no value is specified it returns the current mark type, otherwise no result

Description

The **-marked** attribute tells the handle to take in consideration only messages that are marked with the **markType** postfix. The valid values for markType are MSG_CLAIMED, MSG_TCIS, MSG_TCIS2, and MSG_EWNL. If this attribute is not specified, then all messages will be considered.

> **NOTE —** The **open** operation will always load messages of all/any mark types.

Example

This positions the handle to the first TCIS2 converted messages.

```
> $MSG_conn -marked MSG_TCIS2 -first
```

Exceptions

Invalid attribute value.

14.1.2.7 -maxMsgs

Syntax

xmyMsgDir open -maxMsgs number

Returns

No result.

Description

The **-maxMsgs** attribute is used to control the number of messages (file names) in the Message Response Directory which are loaded into memory when the handle is created. The messages are loaded starting with the newest messages. If this attribute is not specified or if the value of 0 is specified, **all** messages in the directory will be loaded. The **-numMsgs** attribute can be used to return the actual number of messages loaded.

Example

This loads the first (newest) 200 messages received by the SOAC1 TOPCOM handler.

```
> xmyMsgDir open -topcom SOAC1 -maxMsgs 200
```

Exceptions

Invalid attribute value

14.1.2.8 -move

Syntax

handleName -move ?*time?*

Returns

When no value is specified it returns the current “move time value”, otherwise no result

Description

The **-move** attribute computes the “move time value” by adding a positive attribute value or subtracting a negative value from the handle’s current time position. The move attribute value is a time expressed in seconds. The handle’s current position is set to the first message in the Message Response Directory which arrived ***after*** the computed “move time value”.

Example

This gets the next message that arrived five minutes after the current message.

```
> $MSG_conn -move 300 -data
```

Exceptions

- The current time position in the Message Response Directory has not be set.
- No message can be found after the computed “move time value”
- Invalid attribute value

14.1.2.9 -msgDir

Syntax

```
handleName -msgDir
```

Returns

The path to the Message Response Directory associated with the handle

Description

The **-msgDir** attribute returns the Message Response Directory that was opened and loaded when the **xmyMsgDir** handle was created.

Example

```
> $MSG_conn -msgDir
```

14.1.2.10 -next

Syntax

handleName `-next`

Returns

No result

Description

Based on the **-marked** attribute value, the **-next** attribute sets the handle's current time position to the next received (newer) message loaded from the Message Response Directory.

Example

This gets the file name of the next received EWNL converted message.

```
> $MSG_conn -marked MSG_EWNL
> $MSG_conn -next
> set MSG_file [$MSG_conn -file]
```

Exceptions

- No next message is found
- The current time position in the Message Response Directory has not be set.

14.1.2.11 -numMsgs

Syntax

```
handleName -numMsgs
```

Returns

The number of messages loaded from the Message Response Directory

Description

The **-numMsgs** attribute returns the number of messages (file names) loaded into memory from the Message Response Directory when the **xmyMsgDir** instance handle was created using the `open` attribute. The **-maxMsgs** attribute can be used to control the number of messages loaded.

Example

```
> $MSG_conn -numMsgs
```

14.1.2.12 -position

Syntax

handleName -position *?time?*

Returns

The handle's current time position in the Message Response Directory when no value is specified, otherwise no result.

Description

The **-position** attribute sets the handle's current time position to the first message in the Message Response Directory which arrived ***after*** the given position attribute value. The position value is the time in UNIX format (i.e. seconds elapsed since 1/1/1970). When no value is specified the current time position is returned.

Example

This returns the TCIS converted message that arrived after time 89234234.

```
> $MSG_conn -marked MSG_TCIS
> $MSG_conn -position 89234234 -data
```

This returns the time position of the current message.

```
> $MSG_conn -position
```

Exceptions

- No message can be found after the given time position
- Invalid time position value

14.1.2.13 -prev

Syntax

handleName -prev

Returns

No result

Description

The **-prev** attribute sets the handle's current time position to the previous received (older) message in the Message Response Directory.

Example

This returns get the time stamp of the previous message.

```
> $MSG_conn -prev
> $MSG_conn -position
```

Exceptions

- No previous message is found
- The current time position in the Message Response Directory has not be set.

14.1.2.14 -printcom

Syntax

```
xmyMsgDir open -printcom printcomHandlerName
```

Returns

No result. This attribute can only be specified with the **open** method

Description

The **-printcom** attribute is set on a per handle basis and is set only through the **xmyMsgDir open** class method. Given the ***printcomHandlerName***, a search is performed in the configuration file to determine the location (Message Response Directory) of the received messages to load. Since the PRINTCOM protocol does not use session numbers, the default value is 0. The ***printcomHandlerName*** is the same value that is required for the **xmyPrt3270 connect** class method

Example

This creates a handle to get messages from the PRT3270 handle named **printer_1**.

```
> set handle [xmyMsgDir open -printcom printer_1]
```

Exceptions

The specified printcom handler does not exist in the configuration file

14.1.2.15 -recvSession

Syntax

```
handleName -recvSession
```

Returns

The receive session number associated with the messages in the Message Response Directory

Description

The **-recvSession** attribute returns the session number which the messages were received on. The **xmyMsgDir** instance handle was created with a given protocol handler name and each protocol handler entry in the configuration file has a receive session number.

Example

```
> $MSG_conn -recvSession
```

14.1.2.16 -subDir

Syntax

```
xmyMsgDir connect -subDir subDir
```

Returns

The connection handle.

Description

The **-subDir** attribute allows the specifying of a sub directory. The messages are stored in a separate directory for each configured AppApp protocol handler, for example, *$XMYHOME/data/messages/AppApp/app_1*. The given sub directory is appended to this path.

> **NOTE —** This option is not valid for TOPCOM or PrintCom handlers.

-subDir is useful for cases where connection oriented protocol is used for sending and receiving messages. Connection oriented means not using the **-broadcast** attribute. In such cases, the messages are stored in a sub directory identified by the connection id.

Example

```
> xmyMsgDir connect -subDir conn_1 -handler app_1
```

14.1.2.17 -topcom

Syntax

```
xmyMsgDir open -topcom topcomHandlerName
```

Returns

No result. This attribute can only be specified with the **open** method

Description

The **-topcom** attribute isset on a per handle basis and is set only through the **xmyMsgDir open** class method. Given the ***topcomHandlerName***, a search is performed in the configuration file to determine the location (Message Response Directory) and session number of the received messages to load. The ***topcomHandlerName*** is the same value that is required for the **xmyTop connect** class method

Example

This create a handle to scan messages received by the TOPCOM handle named **handler_1**.

```
> set MSG_conn [xmyMsgDir open -topcom handler_1]
```

Exceptions

The specified topcom handler does not exist in the configuration file

14.1.3 Example

This example scans the first 250 messages received by TOPCOM handler **LFACS_1** looking for the newest TCIS2 converted message that contains the tag “SEQ” with the value of “4579” (i.e., SEQ=4579).

```
# Load the newest 250 LFACS_1 messages
set dirHandle [xmyMsgDir open -topcom “LFACS_1” -maxMsgs 250]

# Interested in only the TCIS messages
$dirHandle -marked MSG_TCIS2

# Set the handle to the newest (first) TCIS2 message
$dirHandle -first

# Loop through the TCIS2 messages from newest to oldest
# looking for “SEQ=4579
while { [regexp “SEQ=4579” [$dirHandle -data]] != 1} {

  # Get the next newest TCIS message. If none are left
  # catch the exception
  if { [catch {$dirHandle -prev} result] == 1 } {

     # No messages left
     puts “Didn’t find the TCIS messgae containing SEQ=4579”
     $dirHandle close
     xmyExit
  }
}

# Found the message
puts “Found the mesage : [$dirHandle -data]”

$dirHandle close
```

The lines

```
set dirHandle [xmyMsgDir open -topcom “LFACS_1” -maxMsgs 250]
$dirHandle -marked MSG_TCIS2
```

can be combined into one line, as in

```
set dirHandle [xmyMsgDir open -topcom “LFACS_1” -maxMsgs 250\
     -marked MSG_TCIS2]
```

14.2 Match Tcl Extensions

The Application to Application Match extensions are used to determine whether received messages match specified criteria.

14.2.1 xmyMsgMatch

Syntax

`xmyMsgMatch` *?timeout? ?handle matchTclProc?*

Returns

1 if a message was received and matched, 0 otherwise

Description

The **xmyMsgMatch** procedure is used to receive those messages from a connection that satisfy the user defined match criteria(s). The match criteria is defined within a Tcl procedure. The **xmyMsgMatch** procedure takes three arguments and will return 1 only if the given **matchTclProc** matches a message received over the given connection `handle` within the given timeout period.

The **matchTclProc** is the name of a user defined Tcl procedure which must return 1 if the message satisfies the match criteria and 0 otherwise. This user defined Tcl procedure must take the handle name as its only argument. The handle name is used to access the current received message to determine if that message matches the user criteria(s). Since the **matchTclProc** is called with each received message, it should be developed with performance considerations.

If **-1** is specified as the **xmyMsgMatch timeout** value, then the **$handle -timeout** value will be used.

timeout is and integer specifying the number of seconds to wait for the desired message.

Example

```
> # Receive the printer message that
> # Contains the text “Page 1”
> # Define the User Tcl procedure to match this message
> proc matchMyPrtResp { msgHandle } {
  return [regexp “Page 1” [$msgHandle -data]]
}
```

```
> # Create the connection to get the handle
> set msgHandle [xmyPrt3270 connect -printcom “myPrinter”]
> # Wait up to 60 seconds for the desired message
> if {[xmyMsgMatch 60 $msgHandle matchMyPrtResp] == 1} {
      # This is the file containing the desired message
      set myMsg [$msgHandle -file]
}
> $msgHandle disconnect
```

Exceptions

- Missing or invalid arguments
- The user defined **matchTclProc** contains an error
- Exceptions associated to the handle’s **receive** method

14.2.2 xmyMsgMatchUntil

Syntax

```
xmyMsgMatchUntil timeout maxMsgs handle firstMatchTclProc \
                 untilMatchTclProc1 [untilMatchTclProc2 ...]
```

Returns

1 if a message was received and matched, 0 otherwise

Description

The **xmyMsgMatchUntil** procedure is used to receive a message that is sent in multiple pieces with known starting and ending pieces. The first (starting) piece of the message begins with the first received message which satisfies the match criteria in **firstMatchTclProc**. All subsequent received message are appended to the starting piece until all the **untilMatchTclProcs** are satisfied in the listed order. At least one **untilMatchProc** must be specified to match on the ending piece.

The **maxMsgs** argument value limits the number of messages appended together in the event **untilMatchTclProcs** never matches on an ending message. If the `maxMsgs` and **timeout** values are **-1**, then the **$handle -maxMsgs** and **$handle -timeout** values are used, respectively.

The **firstMatchTclProc** and **untilMatchTclProcs** are Tcl procedures provided by the user and they must return 1 if the message matches and 0 otherwise. These procedures must take the handle name as the only argument. The handle name is used to access the received message to determine if that message meets the match criteria. Since **firstMatchTclProc** is called with each initial message and the **untilMatchTclProcs** are called with subsequent messages, they should be developed with performance considerations.

Example

```
# Match Pages 2 through 7 of a printer message assuming
# each page arrives as a separate printer message

# Define the Tcl procedure to match on the starting piece
# which contains the text “Page 2”

> proc matchFirst { handle } {
      return [regexp ”Page 2” [$handle -data]]
}

# Define the Tcl procedure to match on the ending piece
# which contains the text “Page 7”

> proc matchUntil { handle } {
      return [regexp ”Page 7” [$handle -data]]
}
```

```
# Create printer connection and get the handle

> set handle [xmyPrt3270 connect -printcom “myPrinter”]

# wait up to 60 seconds to receive the message consisting of
# multiple received pieces (pages 2 through 7)

> if {[xmyMsgMatchUntil 60 10 $handle matchFirst \
           matchUntil] == 1} {
     # This is the file containing the messages (pages) I want
     set myMsg [$handle -file]
}

$handle disconnect
```

Exceptions

- Missing or invalid arguments
- The user defined Tcl procedures contains an error
- Exceptions associated to the handle’s `receive` method

14.2.3 xmyMsgMatchNext

Syntax

`xmyMsgMatchNext timeout` *`handle`* `matchTclProc` *`number`*

Returns

1 if a message was received and matched, 0 otherwise

Description

The **xmyMsgMatchNext** procedure is used to receive a message that is sent in multiple pieces with a known starting piece and number of subsequent pieces. The first piece of the message is the first received message that satisfies the match criteria defined in the **MatchTclProc**. The next `number` of received messages will be appended to the first piece.

The **matchTclProc** is the name of a user defined Tcl procedure which must return 1 if the message satisfies the match criteria and 0 otherwise. This user defined Tcl procedure must take the handle name as its only argument. The handle name is used to access the current received message to determine if that message matches the user criteria(s). Since the **matchTclProc** is called with each received message, it should be developed with performance considerations.

If **-1** is specified as the **xmyMsgMatch timeout** value, then the **$handle -timeout** value will be used.

Example

```
# Receive the first 5 pages of a printer job assuming each
# page arrives as a separate message
# Define the User Tcl procedure to match page 1
> proc matchMyPrtResp { handle } {
  return [regexp “Page 1” [$handle -data]]
}
# Create the connection to get the handle
set handle [xmyPrt3270 connect -printcom “myPrinter”]
# Wait up to 60 seconds for the desired message
> if {[xmyMsgMatchNext 60 $handle matchMyPrtResp 4] == 1} {
      # This is the file containing pages 1 to 5
      set myMsg [$handle -file]
}
$handle disconnect
```

Exceptions

- Missing or invalid arguments
- The user defined Tcl procedures contains an error
- Exceptions associated with the handle’s **receive** method

14.3 Marking/Unmarking Messages - xmyMsgMarkFile

Syntax

```
xmyMsgMarkFile fileName markType ?flag?
```

Returns

No result when `flag` is specified otherwise 1 if the message is marked and 0 if the message is not marked.

Description

The **xmyMsgMarkFile** command is used to mark/unmark a message in the Message Response Directory. The message is identified by the name of the file containing it. The **markType** argument specifies the type of mark and the only valid value is currently MSG_CLAIMED. To mark the message the flag has to be 1, to unmark the message the flag has to be 0. If no flag is provided the command returns the current status of the file: 1 for marked and 0 for not marked.

Example

Mark as claimed the last message that was received.

```
> set mark_file[$MSG_conn -file]
> xmyMsgMarkFile $mark_file APP_MSG_CLAIMED 1
```

Exceptions

- Filename doesn’t exist
- Filename is invalid

15. Batch Tcl Language Extensions

This system design describes MYNAH tools that permit the user to

1. Submit JCL, stored in a UNIX file, to a user-selected MVS host machine for batch execution
2. Optionally edit the JCL to dynamically change JCL parameters
3. Query the host system to determine the status of that batch job
4. Retrieve the JES output of batch jobs
5. Retrieve batch job condition codes
6. Delete batch jobs from the output queue and any temporary files that were created by these tools on Unix.

The designed solution provides batch submission via Tcl.

> **NOTE —** The MYNAH Batch Tcl extensions are implemented as Tcl procedures.

15.1 Accessing The Batch Procedures

Access to the batch procedure will be provided via the SE. You do not need to use the Tcl **source** statement to access the batch procedure.

15.2 Submitting a Batch Job - batch_submit

Syntax

```
batch_submit -file jclfile -host hostname \
      -starting_string string -ending_string string \
      ?-edit {from_string to_string}? ?-timeout seconds?
```

Returns

A Tcl keyed list. This keyed list must be saved to a variable for subsequent batch procedures.

On failure an exception is thrown.

Description

The **batch_submit** procedure is used to submit a single batch job for processing. **batch_submit** attempts to submit the JCL contained in the user-specified file to the user-specified host after optionally editing the JCL if the user provided edits.

batch_submit returns a Tcl keyed list that subsequent MYNAH automated batch job procedures require to perform processing on the batch job. The keyed list returned from each successful batch job submitted via **batch_submit** ***must*** be stored into its own variable. The keyed list is used later to determine if the job was submitted successfully, to determine the status of the batch job (i.e., the job is in the input queue, the job is active, or the job is in the JES output queue), to retrieve the JES output of the batch job, and to retrieve individual batch job step results.

More than one batch job may be submitted from a single Tcl script such that batch jobs can execute in parallel on the MVS machine, provided system resources are available and the batch job names are different. Note that a job name is an MVS user id plus an optional suffix. The suffix may be changed to submit multiple jobs to run in parallel.

batch_submit takes the following options:

-file ***jclfile***	The name of the *required* file that contains the JCL to be processed be passed on the command line. The full path to the file must be specified.
-host ***hostname***	The name of the host machine where the JCL will be processed. The host machine name must be in the *.netrc* file of the UNIX machine submitting the batch job.
-starting_string ***string***	The ***starting*** string is a unique string that is found on the line immediately proceeding the job STEP information. **batch_submit** *requires* a starting string and ending string to locate STEP results in the JES output. The starting and ending strings must not contain a hyphen.
-ending_string ***string***	The ***ending*** string is a unique string that is found on the line immediately after the job STEP information. **batch_submit** *requires* a starting string and ending string to locate STEP results in the JES output. The starting and ending strings must not contain a hyphen.
-edit ***{from_string to_string}***	This identifies a two element Tcl list containing ***from*** and ***to*** strings. The ***from*** string will be searched for in the JCL and, if found, replaced by the ***to*** string in a copy of the user's JCL. The **-edit** attribute and its associated ***from*** and ***to*** strings are optional. Edits are in the form **-edit** {***from to***}. If more than one edit is necessary, the **-edit** attribute and associated two element list may be repeated.

-timeout ***seconds*** This identifies the time limit that a job will be waited for when the **batch_wait** procedure is executed. The **-timeout** attribute and its associated value in seconds are optional. If a timeout value is not specified a default value of 600 seconds is used.

Example

The following Tcl script statement demonstrates submitting a JCL stored in a UNIX file called */u/jcl* to a host machine called **pyibm2**, the editing of a copy of the JCL to change the programmer name from <*PGRM*> to *LP*, the job name from <*JOB*> to *TKTEE66*, and the job class from <*CLASS*> to *X*, and storing of the keyed list into a variable called **job_list**:

```
> set job_list [ batch_submit -file /u/jcl \
     -host pyibm2 \
     -starting_string {BELLCORE JOB SUMM} \
     -ending_string -edit
     {“<JOB>”} “TKTEEGG”{ENDED} \
     -edit {"<NAME>" "LP"} -edit {"<CLASS>" "X"} ]
```

The JCL stored in the file called */u/jcl* for the above example Tcl script statement could have appeared as follows:

```
//<JOB>X  JOB (5,T531),’<PGRM>’,CLASS=<CLASS>,MSGCLASS=F
/*ROUTE     PRINT RRCDP1
//STEP01    EXEC PGM=IEBGENER
//SYSPRINT  DD SYSOUT=*
//SYSUT1DD  *
HI WORLD
//SYSUT2    DD SYSOUT=(*)
//SYSIN     DD DUMMY
```

NOTE — In the above, **MSGCLASS** must be set to *F* to route output to the buffer.

The preceding **batch_submit** edits the user-provided JCL and places the edited output into a temporary file under */tmp*. The original JCL file is ***not*** changed.

The host name must be a name that exists in the *.netrc* file of the UNIX machine submitting the batch job. (See Section 15.4 for information on the *.netrc* file. A suffix may be added.

The editing performed by the preceding **batch_submit** in the example Tcl script statement would change the programmer name from <*PGRM*> to *LP*, the job name from <*JOB*> to *TKTEE66*, and the job class from <*CLASS*> to *X* for previously shown sample JCL.

batch_submit submits the edited JCL by logging onto the user-specified MVS host machine via the **ftp** command and submitting the JCL.

Exceptions

This method will fail (throw an exception) for the following conditions:

- Problem with **-file**
  - — **-file** must have a value
  - — **-file *file_name*** does not exist
  - — **-file *file_name*** is not readable
  - — **-file *file_name*** is empty
- Problem with **-host**
  - — **-host** must have a value
  - — environment variable HOME not defined
  - — file *$HOME*/.netrc does not exist
  - — file *$HOME*/.netrc is not readable
  - — file *$HOME*/.netrc is empty
  - — host *hostname* not in *$HOME/.netrc*
- Problem with **-edit**

  Edit must be of the form **-edit** {*from_string to_string*}
- Problem with **-starting_string**

  -starting_string must have a non-null value
- Problem with **-ending_string**

  -ending_string must have a non-null value
- Problem with **-timeout**

  -timeout must have positive integer value
- The **ftp** command failed.

  ```
  ftp command failed
  ```
- Incorrect usage

  ```
  Usage = batch_submit -file jclfile\
        -host hostname -starting_string string \
        -ending_string string ?-timeout seconds? \
        ?-edit {from to}?
  ```

15.3 Methods

Just as the other MYNAH extension packages use methods (sub-commands to class commands that perform actions on handles you create) the Batch package has procedures that perform actions on the Tcl keyed lists you create using **batch_submit**. For the sake of continuity, we will also refer to these procedures as methods as well as procedures.

The general syntax of these methods/procedures is

```
procedure job_list arguments
```

15.3.1 batch_delete

Syntax

```
batch_delete job_list
```

Description

The **batch_delete** procedure/method lets you free spooled files associated with a batch job. Its important to remove the spooled files to save space and to improve the performance of the MYNAH batch processing since the **ftp** command that **batch_submit** and **batch_wait** use are affected by the number of spooled files.

Example

```
> set job_list [ batch_submit -file "/home/my_jcl" \
      -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
      -ending_string {ENDED} -timeout 600 ]
```

This waits up to ten minutes for the batch job to complete.

```
> set job_status [ batch_wait job_list ]
```

Delete the batch job's JES output.

```
> if { $job_status == "OUTPUT" } {
      batch_delete job_list
}
```

Exceptions

batch_delete will handle errors by generating a Tcl exception.

- Missing **job_list**

  No value given for parameter **job_list** to **batch_delete**

- The **ftp** command failed.

  ```
  ftp command failed
  ```

- The JOB was not found

  ```
  JOB NOT FOUND
  ```

- The JOB was already deleted

  ```
  JOB 'job_name' already deleted
  ```

15.3.2 batch_host

Syntax

```
batch_host job_list
```

Returns

```
hostname
```

Description

The **batch_host** procedure/method lets you query for the name of the host machine. **batch_host** requires the keyed list generated by **batch_submit**.

Example

```
> set job_list [ batch_submit -file "/home/my_jcl" \
      -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
      -ending_string {ENDED} -timeout 600 ]
```

Determine the host name of the batch job.

```
> set host_name [ batch_host job_list ]
> puts "host name is $host_name"
```

Exceptions

batch_host will generate a Tcl exception under the following condition:

- Missing **job_list**

  No value given for parameter **job_list** to **batch_host**

15.3.3 batch_jobid

Syntax

`batch_jobid` *`job_list`*

Returns

JOB##### if successful and where ##### is the job id assigned by the host

Description

The **batch_jobid** procedure/method lets you query for the job id assigned to a batch job by the IBM mainframe. **batch_jobid** requires the keyed list generated by **batch_submit**.

Example

```
> set job_list [ batch_submit -file "/home/my_jcl" \
      -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
      -ending_string {ENDED} -timeout 600 ]
```

Determine the name of the batch job.

```
> set job_id [ batch_jobid job_list ]
> puts "the job id is $job_id"
```

Exceptions

batch_jobid will generate a Tcl exception under the following condition:

- Missing **job_list**

  No value given for parameter **job_list** to **batch_jobid**

15.3.4 batch_status

Syntax

`batch_status` *`job_list`*

Returns

`INPUT`, `ACTIVE`, or `OUTPUT`

Description

The **batch_status** procedure/method lets you query the IBM mainframe to determine the status of a batch job. **batch_status** requires the keyed list generated by **batch_submit**.

If a batch job has completed, **batch_status** will retrieve a copy of the JES listing segment of the job and extract the condition code lines from the listing for subsequent batch job analysis.

Example

```
> set job_list [ batch_submit -file "/home/my_jcl" \
      -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
      -ending_string {ENDED} -timeout 600 ]
```

Determine the status of the batch job.

```
> set job_status [ batch_status job_list ]
```

```
> if { $job_status == "INPUT" } {
      puts "Job is queued for execution"
} elseif { $job_status == "ACTIVE" } {
      puts "Job is active"
} else { $job_status == "OUTPUT" } {
      puts "Job is complete"
}
```

Exceptions

batch_status will generate a Tcl exception under the following conditions:

- Missing **job_list**

  No value given for parameter **job_list** to **batch_status**

- The **ftp** command failed.

  `ftp command failed`

15.3.5 batch_step_count

Syntax

```
batch_step_count job_list
```

Returns

Number of steps in batch job

Description

The **batch_step_count** procedure/method lets you query for the number of steps reported in a batch job. **batch_step_count** requires the keyed list generated by **batch_submit**.

Example

```
> set job_list [ batch_submit -file "/home/my_jcl" \
      -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
      -ending_string {ENDED} -timeout 600 ]

Wait up to ten minutes for the batch job to complete

> set job_status [ batch_wait job_list ]
```

Get the results if the job complete and determine the number of steps reported in this batch job.

```
> if { $job_status == "OUTPUT" } {
      set steps = batch_step_count job_list
      puts "this job had $steps reported

}
```

Exceptions

batch_step_count will generate a Tcl exception under the following condition:

- Missing **job_list**

  No value given for parameter **job_list** to **batch_step_count**

15.3.6 batch_step_result

Syntax

`batch_step_result` *`job_list step_number`*

Returns

Result string from JES if step exists

Null string if step does not exist

Description

The **batch_step_result** procedure/method lets you query for the condition code associated with a user-specified ***step number*** of a completed batch job. **batch_step_result** requires the keyed list generated by **batch_submit**.

batch_step_result can only obtain step results after a **batch_status** or **batch_wait** has indicated that the batch job has completed. The following are examples of step results that might be returned:

```
CONDITION CODE: 0000 IEBGENER
SYSTEM ABEND: 0806 IEBGENER
```

Example

```
set job_list [ batch_submit -file "/home/my_jcl" \
     -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
     -ending_string {ENDED} -timeout 600 ]
#
# wait up to 10 minutes for the batch job to complete
set job_status [ batch_status job_list ]
#
# get the results if the job completed
#
if { $job_status == "OUTPUT" } {
#
# determine the number of steps in the batch job
#
     set step_count [ batch_step_result job_list ]
#
# get the step result of each step of the job
#
     for { set step 1 } { $step <= step_count} \
          { incr step +1 } {

          set step_result [batch_step_result job_list $step]

          if { $step_result == "" } {
               puts "$step, No results"
          } else {
               puts "$step, $step_result"
          }

     }

}
```

Exceptions

batch_step_result will generate a Tcl exception under the following conditions:

- Missing **job_list**

  No value given for parameter **job_list** to **batch_step_result**

- The batch job has not completed.

  `batch job ‘job’ status is ‘status’`

- The step number is not found

  `Key “<step#>” is not found in the keyed list`

15.3.7 batch_wait

Syntax

`batch_wait` *`job_list`*

Returns

`INPUT`, `ACTIVE`, or `OUTPUT`

Description

The **batch_wait** procedure/method lets you wait up to a user-defined time in seconds for a batch job to reach a job status of **OUTPUT**. If the batch job completes prior to the user-defined wait time, **batch_wait** will return with a result of **OUTPUT** prior to the wait time elapsing. If the batch job does not complete prior to the user-defined wait time, the method will return the current status of the job after the user-defined time has elapsed. If the user does not supply a time to wait, a default of 0 seconds will be used.

If a batch job has completed, **batch_wait** will retrieve a copy of the JES listing segment of the job and extract the condition code lines from the listing for subsequent batch job analysis. **batch_wait** uses the starting string and ending string defined by **batch_submit** to locate the condition codes.

Example

```
set job_list [ batch_submit -file "/home/my_jcl" \
     -host "pyibm2" -starting_string {BELLCORE JOB SUMM} \
     -ending_string {ENDED} -timeout 600 ]
#
# wait up to 10 minutes for the batch job to complete
#
set job_status [ batch_wait job_list ]
#
# what was the status
#
if { $job_status == "INPUT" } {
     puts "Job is queued for execution"
} elseif { $job_status == "ACTIVE" } {
     puts "Job is active"
} elseif { $job_status == "OUTPUT" } {
     puts "Job is complete"
}
```

Exceptions

batch_wait will generate a Tcl exception under the following conditions:

- Missing **job_list**

  No value given for parameter **job_list** to **batch_step_count**

- The **ftp** command failed.

  ```
  ftp command failed
  ```

15.4 The .netrc file

The SE used to process batch jobs must have a *.netrc* file in its $HOME directory. The *.netrc* file must have permissions of 0600 (read and write by owner) for the user-id executing the SE.

The *.netrc* file must contain the following for each machine and login to be used:

```
machine <machine_name> login <login> password <password>
```

16. DCE Extension Package

This DCE Extension Package provides the functions necessary for interactions with a Distributed Computing Environment (DCE) client or server. Unlike the other MYNAH packages, the DCE scripting language is not available directly from MYNAH. Instead, DCE scripting is available from an external application-specific emulated client or server executable, which the MYNAH Administrator must build, as described in Section 11 of the *MYNAH System Administration Guide*.

Users may use the emulated client or server independently of MYNAH, or they may automate their use from MYNAH with the ASYNC package. The MYNAH installation includes several Tcl routines that use the ASYNC package to automate the execution of DCE scripts and verify their outputs (e.g., **xmyDceStartClient**, **xmyDceStartServer**. See Section 16.4.).

The documentation in this chapter describes the DCE scripting language available in the emulated client and server executables.

Table 16-1 contains brief descriptions of the contents of each subsection in this section.

Table 16-1. MYNAH DCE Extension Sections

Section Name	Description	Section Number
DCE Overview	This section contains a basic discussion on DCE.	16.1, Page 16–2
Overview of Scripting	This section provides basics on creating DCE scripts.	16.3, Page 16–4
Using the Emulated Client and Emulated Server in MYNAH System	This section describes how to use the emulated client and emulated server executables to test DCE applications.	16.4, Page 16–5
Interface Object	This section contains the Tcl extensions for working with an interface object.	16.5, Page 16–11
IDL Types	This section contains the Tcl extensions for working with Interface Description Language (IDL) files.	16.6, Page 16–14
RPC Calls in the Emulated Client	This section contains scripting information on working with RPC Calls in the Emulated Client.	16.7, Page 16–69
RPC Calls in the Emulated Server	This section contains scripting information on working with RPC Calls in the Emulated Server.	16.10, Page 16–72

Table 16-1. MYNAH DCE Extension Sections

Section Name	Description	Section Number
Constants	This section contains information on the Tcl created constant for the interface.	16.11, Page 16–73
Destroying Objects	This section contains information on destroying interface objects.	16.12, Page 16–74
Deleting Handles and Objects	This section contains information on deleting DCE handles.	16.13, Page 16–75

16.1 DCE Overview

This section provides you with a high-level understanding of DCE. It is not meant to be a complete tutorial on DCE.

16.1.1 DCE Architecture

DCE provides distributed application programming in the form of the **client-server** model. In this model, a **server** is an application that provides a **service**. A **client**, on the other hand, is an application that requests a server to perform some service on its behalf.

Specifically, DCE uses the **Remote Procedure Call** (RPC) mechanism to accomplish client-server processing. Using RPCs, the client calls a procedure that appears to be a local procedure. In reality, the procedure that is really invoked is one that is defined in a server application.

The RPCs that a client can call (and that a server must provide) are defined in the interface.

16.1.2 Interface Definition

The interface provides the cohesion between client and server, defining the operations that a client may call and that a server must provide. Since an operation may take data as input and may produce data as output, the interface must also define the what this data looks like.

The interface itself is defined by a set of **Interface Description Language** (IDL) files. Collectively, the IDL files define the operations the client may call and all data types into and out of the operations.

16.1.3 IDL File

An IDL file resembles a C header file. In fact, the syntax of the IDL file is based on the syntax for C declarations.

Within the IDL file you will find type definitions and operation declarations. Type definitions resemble the C **typedef** syntax. Operation declarations resemble ANSI C procedure declarations with the addition of annotations on each procedure parameter. These annotations determine the data flow (client to server, server to client, or both) for each parameter.

16.2 Developing a DCE Application

The steps in developing an application depend on whether the application will be a client or a server. Both routes start the same: the interface must be defined.

16.2.1 DCE Client Development

Once the interface is defined, the interface must be compiled (using the **idl** tool) to produce client stubs and an interface header file. These stubs define local procedures that resemble the operations defined in the interface. When the client application calls one of these local procedures, the DCE library forwards the request to the server for execution of the actual procedure.

The interface header file is `#include`'d by other source files in the client application. The header file provides C prototypes for the operations defined in the interface, as well as C versions of the type definitions.

The client application must be linked with the generated client stubs and with the DCE runtime libraries.

16.2.2 DCE Server Development

In the server side, the interface must also be compiled. This time it produces server stubs in addition to the interface header file. These stubs provide the necessary hooks so that as a request comes in, DCE can invoke the correct server procedure.

The developer must implement the RPC operations defined in the interface. This entails writing C procedures for each operation.

Analogous to the client application, the server application must be linked with the generated server stubs and the DCE runtime library.

16.3 Overview of Scripting

Before you begin, the MYNAH adminstrator must build emulated clients and servers.

The emulated client is essentially an interactive Tcl interpreter with additional commands in support of the DCE domain.

The emulated server is also an interactive Tcl interpreter, but it is more complex than the emulated client due to the fact that DCE is in control instead of the Tcl interpreter. DCE only turns control over to Tcl temporarily when a DCE request comes in.

Once the emulated clients and servers have been built, you can then use tem to test DCE applications. Section 16.4 describe the commands used to start the emulated clients and servers.

In the following language overview, the following conventions will be used:

hData	Handle to any data item
hStruct	Handle to a structure
hUnion	Handle to a union
hEnum	Handle to an enum
hInterface	Handle to an interface
hPointer	Handle to a pointer.

16.3.1 Emulated Client

Using an emulated client, you interact with a real DCE server by

1. Connecting to a server
2. Creating input data
3. Creating holders for output data
4. Calling the rpc
5. Verifying output values.

16.3.2 Emulated Server

Using an emulated server, you interact with a real DCE client by

1. Starting the emulated server, providing a script containing definitions for all RPCs
2. Exercising client application (using other domain extensions)

16.4 Using the Emulated Client and Emulated Server in MYNAH System

This section describes how to use the emulated client and emulated server executables to test DCE applications within in the MYNAH System framework.

16.4.1 Overview

Using the methods presented below, you may write MYNAH scripts that start an emulated client or emulated server process, and capture all output from each operation that is executed. This captured output is compared against a baselined image, which updates the compares for the MYNAH script.

16.4.2 Using the Emulated Client

The following methods are used to start an emulated client.

16.4.2.1 xmyDceStartClient

```
xmyDceStartClient \
     -user dce-user \
     -password dce-password \
     -dir directory \
     -entry CDS-entry \
     -script client-script
```

Returns

The asynchronous connection handle on which the DCE emulated client is run

Description

The **xmyDceStartClient** method first logs into DCE using ***dce-user*** and ***dce-password***. It then starts the emulated client executable found in *directory/client.* The client uses the CDS entry ***CDS-entry*** for communicating with the server. The client runs the DCE script ***directory/client-script***.

The name of each output file is *dce.client.opname.sequence-number*, where ***opname*** is the name of the operation and ***sequence-number*** is incremented for each operation that is executed. The baseline images used for comparison are found in the directory *directory/baselined.images.*

xmyDceStartClient should be followed by the **xmyDceWaitForClient** command.

Exceptions

Asynchronous connection exceptions

Example

```
> set clientConn [xmyDceStartClient \
      -user "ksb" \
      -password "mm_dce" \
      -dir "/u/dce-stuff/tcl-dce/unit-tests/testing"\
      -entry "/.:/test/locnet/empty-g" \
      -script "empty-client-stubs.tcl" \
      ]
```

16.4.2.2 xmyDceWaitForClient

```
xmyDceWaitForClient connectionHandle
```

Returns

An empty string

Description

The **xmyDceWaitForClient** method waits for output from the emulated client running on ***connectionHandle***. As it receives output, it compares the new output against the baselined output in *directory/baselined.images*. The comparison is made using the **xmyDiff** command, which updates the compares for the running MYNAH script.

Exceptions

Asynchronous connection exceptions

Example

```
> xmyDceWaitForClient $clientConn
```

16.4.3 Using the Emulated Server

The following methods are used to start an emulated server.

16.4.3.1 xmyDceStartServer

```
xmyDceStartServer \
      -user dce-user \
      -password dce-password \
      -dir directory \
      -entry CDS-entry \
      -script server-script \
      ?-timeout timeout?
```

Returns

The asynchronous connection handle on which the DCE emulated server is run

Description

The **xmyDceStartServer** method first logs into DCE using ***dce-user*** and ***dce-password***. Then it starts the emulated server executable found in *directory/server*. The server creates an entry in the group entry at ***CDS-entry*** for communicating with the client; it also creates an entry for the server using ***CDS-entry****.host.pid*. The server runs the DCE script ***directory/server-script***. If the timeout parameter is provided, and is non-zero, it represents the amount of time (in minutes) that the server should be allowed to run.

The name of each output file is *dce.server.opname.sequence-number*, where ***opname*** is the name of the operation and ***sequence-number*** is incremented for each operation that is executed. The baseline images used for comparison are found in the directory *directory/baselined.images*.

xmyDceStartServer should be followed by the **xmyDceWaitForServer** command.

Exceptions

Asynchronous connection exceptions

Example

```
> set serverConn [xmyDceStartServer \
      -user "ksb" \
      -password "mm_dce" \
      -dir "/u/dce-stuff/tcl-dce/unit-tests/testing" \
      -entry "/.:/test/locnet/empty-g" \
      -script "empty-server-stubs.tcl" \
      -timeout 5 \
      ]
```

16.4.3.2 xmyDceWaitForServer

```
xmyDceWaitForServer connectionHandle
```

Returns

An empty string

Description

The **xmyDceWaitForServer** method waits for output from the emulated server running on ***connectionHandle***. As it receives output, it compares the new output against the baselined output in *directory/baselined.images*. The comparison is made using the **xmyDiff** command, which updates the compares for the running MYNAH script.

Exceptions

Asynchronous connection exceptions

Example

```
> xmyDceWaitForServer $serverConn
```

16.4.4 Using the Emulated Server for Starting a Long-Running Server

Test scripts may use the **xmyDceStartIndependentServer** method to bring up a long-running emulated server process. This method does not perform any comparisons on the outputs.

16.4.4.1 xmyDceStartIndependentServer

```
xmyDceStartIndependentServer \
      -user dce-user \
      -password dce-password \
      -dir directory \
      -entry CDS-entry \
      -script server-script \
      -log logFile
```

Returns

The process id of the emulated server process

Description

The **xmyDceStartIndependentServer** method behaves like **xmyDceStartServer**, however, it creates a long-running server that runs in the background. Therefore, no comparisons are made against baselined outputs. Instead, all output from the server executable is dumped into the log file given by *logFile*.

Exceptions

Asynchronous connection exceptions

Example

```
> xmyDceStartIndependentServer \
      -user "ksb" \
      -password "mm_dce" \
      -dir "/u/dce-stuff/tcl-dce/unit-tests/testing" \
      -entry "/.:/test/locnet/empty-g" \
      -script "empty-server-stubs.tcl" \
      -log "iserver.log"
```

16.5 Interface Object

The interface is available at runtime through a handle. The name of the handle is based on the name of the interface: **interface-*name***. This handle may be used to determine information about the interface at script execution time.

There is only one method, **info**, which takes an argument noting the information that is being queried.

16.5.1 name

```
hInterface info name
```

Returns

The name of the interface

Description

The **name** argument returns the name of the interface from the interface definition.

16.5.2 uuid

```
hInterface info uuid
```

Returns

The UUID of the interface

Description

The **uuid** argument returns the unique identifier of the interface from the interface definition.

16.5.3 major-version

```
hInterface info major-version
```

Returns

The major version number of the interface

Description

The **major-version** argument returns the major version of the interface from the interface definition.

16.5.4 minor-version

```
hInterface info minor-version
```

Returns

the minor version number of the interface

Description

The **minor-version** argument returns the minor version of the interface from the interface definition.

16.5.5 isClient

```
hInterface info isClient
```

Returns

1 if the interface is in an emulated client, 0 otherwise

Description

The **isClient** argument determines at runtime if the interface is running in an emulated client or emulated server.

16.5.6 isServer

```
hInterface info isServer
```

Returns

1 if the interface is in an emulated server, 0 otherwise

Description

The **isServer** argument determines at runtime if the interface is running in an emulated client or emulated server.

16.5.7 constants

```
hInterface info constants
```

Returns

The list of all constants defined in the interface

Description

The **constants** argument returns a list of all constants defined in the interface.

16.5.8 types

```
hInterface info types
```

Returns

The list of all types defined in the interface

Description

The **constants** argument returns a list of all types defined in the interface.

16.5.9 rpcs

```
hInterface info rpcs
```

Returns

The list of all operations defined in the interface

Description

The **rpcs** argument returns a list of all operations defined in the interface.

16.6 IDL Types

There are three categories of IDL types methods in the DCE Package:

Intrinsic — These are data instances that are the basic building blocks, the foundation pieces on which the IDLs are created.

Aggregates — These are data instances that contain other data instances. Their bindings mimic their underlying data layout; the binding contains other bindings. These are all are user-defined types. They can be implemented by using **array** and **pointer** types. Since these constructs are used frequently and would require more Tcl-level manipulations, the fake types were created to mask some of the hassle involved in accessing the types.

DCE Types — These are data instances that do not exist as independent types in the DCE specification. The generic interface for these types are presented later. The specific interface depends on the definition of the user-defined type.

Table 16-2 lists the DCE IDL type extensions, organizing them into the categories listed above.

Table 16-2. DCE IDL Type Extensions (Sheet 1 of 3)

Category	Method	Description	Section
Intrinsic	**bool**	Creates a boolean object.	16.6.2, Page 16–19
	byte	Creates a byte object.	16.6.4, Page 16–23
	char	Creates a character (ASCII) object.	16.6.5, Page 16–25
	double	Creates a double (64-bit floating point number) object.	16.6.6, Page 16–27
	enumerations	Creates an enumeration object, which contains a set of symbolic constants	16.6.7, Page 16–29
	error_status_t	Creates an error_status_t object, which is an enumerated type used to contain DCE error codes.	16.6.8, Page 16–31
	float	Creates a float object, which is an 32-bit floating point number.	16.6.9, Page 16–33
	handle_t	Creates a handle_t object, which represents a connection to a server.	16.6.10, Page 16–35

Table 16-2. DCE IDL Type Extensions (Sheet 2 of 3)

Category	Method	Description	Section
	hyper	Creates a hyper object, which is an 64-bit signed integer from -2^{63} to $2^{63}-1$.	16.6.11, Page 16–40
	long	Creates a long object, which is a 32-bit signed integer from -2^{31} to $2^{31}-1$.	16.6.12, Page 16–42
	pointer	Creates a pointer object, which is an indirect reference to another object.	16.6.14, Page 16–47
	short	Creates a short object, which is a 16-bit signed integer from -32768 to 32767.	16.6.15, Page 16–50
	small	Creates a small object, which is an 8-bit signed integer from -128 to 127	16.6.16, Page 16–52
	uhyper	Creates an unsigned hyper object, which is an 64-bit unsigned integer from 0 to $2^{64}-1$.	16.6.19, Page 16–58
	ulong	Creates an unsigned long object, which is an 32-bit unsigned integer from 0 to $2^{32}-1$.	16.6.20, Page 16–60
	ushort	Creates an unsigned short object, which is an 16-bit unsigned integer from 0 to 65535.	16.6.22, Page 16–65
	usmall	Creates an unsigned small object, which is an 8-bit unsigned integer from 0 to 255.	16.6.23, Page 16–67
Aggregates	**array**	Creates an array object, which is an ordered series of objects.	16.6.1, Page 16–17
	pipe	Creates a pipe object.	16.6.13, Page 16–44
	structure	Creates a structure object, which is a group of related objects.	16.6.18, Page 16–56
	union	Creates a character object. discriminating unions	16.6.21, Page 16–62

Table 16-2. DCE IDL Type Extensions (Sheet 3 of 3)

Category	Method	Description	Section
Special Types	**buffer**	Creates a buffer object.	16.6.3, Page 16–21
	string	Creates a string object.	16.6.17, Page 16–54

All IDL extensions have a “contstructor” method that creates the handle to the IDL. The constructor takes the form **make-*idltype***, where ***idltype*** is the name of the IDL type.

In addition, the Intrinsic and DCE Types have **set** and **get** methods, which, respectively, let you set and return the values of the handle to the IDL.

16.6.1 array

An array is an ordered sequence of objects. This describes the interface to a generic enumerated type.

There is no method for creating a generic array object. However, if a **typedef** exists defining a new type name for an array (with a fixed size), you can create an instance of that particular array.

16.6.1.1 make-*array* Constructor

Syntax

`make-`*`array`*

Returns

A handle to the newly created array object.

Description

Creates an array object and returns a handle to the object.

Exceptions

Out of memory

16.6.1.2 elements Method

Syntax

`hArray` `elements`

Returns

Handles to all the array elements.

Description

Returns the handles of all the elements in the array.

16.6.1.3 *index* Method

Syntax

hArray index ?method args?

Returns

The result of applying ***method*** to the object at ***index*** in the array.

Description

Invokes ***method*** and ***args*** on the object at the ***index*** position in the array. If ***method*** is not given, returns the handle of the object at that position.

Example

```
> set a [make-xbbMsgOutput_t]
> $a elements
.xmyDceBinding0 .xmyDceBinding1 .xmyDceBinding2
> $e 0 type get
the-type
```

Exceptions

- Index is out of range
- Can’t apply ***method*** and/or ***args*** to element object

16.6.2 bool

A bool is an boolean value. It has two values: **TRUE** and **FALSE**.

16.6.2.1 make-bool Constructor

Syntax

`make-bool` *`?initial-value?`*

Returns

A handle to the newly created bool object.

Description

Creates a bool object and returns a handle to the bool.

Exceptions

- Out of memory
- Illegal initial value

16.6.2.2 get Method

Syntax

`hBool` `get`

Returns

TRUE or **FALSE**

Description

Returns the state of the boolean value as either **TRUE** or **FALSE**.

16.6.2.3 set Method

Syntax

hBool set *newValue*

Returns

Empty string

Description

Sets the value of the bool based on ***newValue***, which can be **1**, **TRUE**, **ON**, or **YES** for TRUE values or **0**, **FALSE**, **OFF**, or **NO** for FALSE values.

Example

```
> set b [make-bool]
> $b set TRUE
> $b get
TRUE
```

Exceptions

Illegal value for ***newValue***

16.6.3 buffer

A buffer is an null-terminated ASCII character sequence, which has a fixed-size array of characters reserved for it.

16.6.3.1 make-buffer Constructor

Syntax

```
make-buffer length ?initial-value?
```

Returns

A handle to the newly created buffer object.

Description

Creates a buffer object and returns a handle to the buffer. The size of the buffer is given by the ***length*** parameter.

Exceptions

Out of memory

16.6.3.2 get Method

Syntax

```
hBuffer get
```

Returns

A string.

Description

Returns the string stored in the buffer.

16.6.3.3 set Method

Syntax

```
hBuffer set newValue
```

Returns

Empty string

Description

Stores ***newValue*** into the buffer object represented by ***hBuffer***. If ***newValue*** is longer than the buffer allows, the string is truncated.

Example

```
> set s [make-buffer 10]
> $s set {abcdefghijklmnop}
> $s get
abcdefgki
```

16.6.3.4 length Method

```
hBuffer length
```

Returns

the length of the buffer

Description

Returns the size of the largest string that can be stored in the buffer.

Example

```
> set s [make-buffer 10]
> $s set {abcdefghijklmnop}
> $s length
10
```

16.6.4 byte

A byte is an 8-bit unsigned integer. It can contain the values from 0 to 255.

16.6.4.1 make-byte Constructor

Syntax

`make-byte` *`?initial-value?`*

Returns

A handle to the newly created byte object.

Description

Creates a byte object and returns a handle to the byte.

Exceptions

- Out of memory
- Illegal initial value

16.6.4.2 get Method

Syntax

`hByte` `get`

Returns

A number between 0 and 255.

Description

Returns the stored number.

16.6.4.3 set Method

Syntax

hByte set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the byte object represented by ***hByte***.

Example

```
> set b [make-byte]
> $b set 255
> $b get
255
```

Exceptions

Illegal value for number

16.6.5 char

A **char** is an 8-bit unsigned integer, like the byte type. It can contain the values from 0 to 255. It is treated as an ASCII character.

16.6.5.1 make-char Constructor

Syntax

`make-char` *`?initial-value?`*

Returns

A handle to the newly created char object.

Description

Creates a char object and returns a handle to the char.

Exceptions

- Out of memory
- Illegal initial value

16.6.5.2 get Method

Syntax

`hChar` `get`

Returns

The ASCII character stored in the char.

Description

Returns the character stored.

16.6.5.3 set Method

Syntax

hChar set *newValue*

Returns

Empty string

Description

Stores *newValue* into the char object represented by *hChar*.

Example

```
> set ch [make-char]
> $ch set c
> $ch get
c
```

Exceptions

Illegal character

16.6.6 double

A double is an 64-bit floating point number.

16.6.6.1 make-double Constructor

Syntax

`make-double` *`?initial-value?`*

Returns

A handle to the newly created double object.

Description

Creates a double object and returns a handle to the double.

Exceptions

- Out of memory
- Illegal initial value

16.6.6.2 get Method

Syntax

`hDouble` `get`

Returns

A floating point number.

Description

Returns the stored number.

16.6.6.3 set Method

Syntax

hDouble set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the double object represented by ***hDouble***.

Example

```
> set f [make-double]
> $f set 3.14159265
> $f get
3.14159265
```

Exceptions

Illegal value for number

16.6.7 enumeration

An enumeration is a type that contains a set of symbolic constants, each associated with a distinct value. This describes the interface to a generic enumerated type.

16.6.7.1 make-*enum* Constructor

Syntax

`make-`*`enum ?initial-value?`*

Returns

A handle to the newly created enumeration object.

Description

Creates an enumerated type object and returns a handle to the object.

Exceptions

- Out of memory
- Illegal initial value

16.6.7.2 get Method

Syntax

`hEnum` `get`

Returns

The symbolic representation of the value.

Description

Returns the value of the enumerated type in symbolic representation.

16.6.7.3 set Method

Syntax

hEnum set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the enumerated type object represented by ***hEnum***.

Example

```
> set e [make-xbbTraceLevel_e]
> $e set idl_TRACE_O
> $e get
idl_TRACE_O
```

Exceptions

Illegal value

16.6.7.4 values Method

Syntax

hEnum values

Returns

The list of all allowed values.

Description

Returns the list of all legal values of the enumerated type in symbolic representation.

16.6.8 error_status_t

An error_status_t is an enumerated type used to contain DCE error codes. Refer to the DCE references for the allowed values. It is implemented as an enumerated type, and supports the standard enumeration methods (see Section 16.6.7).

There are currently 273 valid values for error_status_t. The most important one is **rpc_s_okay**, which is used to represent a successful RPC call.

16.6.8.1 make-error_status_t Constructor

Syntax

```
make-error_status_t ?initial-value?
```

Returns

A handle to the newly created error_status_t object.

Description

Creates an error_status_t object and returns a handle to the object.

Exceptions

- Out of memory
- Illegal initial value

16.6.8.2 get Method

Syntax

```
hErrorStatusT get
```

Returns

The symbolic representation of the value.

Description

Returns the value of the error_status_t in symbolic representation.

16.6.8.3 set Method

```
hErrorStatusT set newValue
```

Returns

Syntax

Empty string

Description

Stores ***newValue*** into the error_status_t object represented by ***hErrorStatusT***.

Example

```
> set e [make-error_status_t]
> $e set rpc_s_not_listening
> $e get
rpc_s_not_listening
```

Exceptions

Illegal value

16.6.8.4 values Method

Syntax

```
hErrorStatusT values
```

Returns

The list of all allowed values.

Description

Returns the list of all legal values of the error_status_t in symbolic representation.

16.6.9 float

A float is an 32-bit floating point number.

16.6.9.1 make-float Constructor

Syntax

```
make-float ?initial-value?
```

Returns

A handle to the newly created float object.

Description

Creates a float object and returns a handle to the float.

Exceptions

- Out of memory
- Illegal initial value

16.6.9.2 get Method

Syntax

```
hFloat get
```

Returns

A floating point number.

Description

Returns the stored number.

16.6.9.3 set Method

Syntax

hFloat set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the float object represented by ***hFloat***.

Example

```
> set f [make-float]
> $f set 3.14159265
> $f get
3.14159265
```

Exceptions

Illegal value for number

16.6.10 handle_t

A handle_t represents a connection to a server. In DCE, this is also known as an **rpc_binding_handle_t**.

16.6.10.1 make-handle_t Constructor

Syntax

```
make-handle_t
```

Returns

A handle to the newly created handle_t object.

Description

Creates a handle_t object and returns a handle to the handle_t object.

Exceptions

Out of memory

16.6.10.2 make uuid_t Constructor

Syntax

```
make-uuid_t ?initialValue?
```

Returns

A handle to the newly created uuid_t object.

Description

Creates a uuid_t object and returns a handle to it. If specified, initialValue is used as the value of the object; otherwise a new UUID value is created.

Exceptions

If initialValue is “” or “NULL”, the value of the object is the NULL UUID.

16.6.10.3 get Method

Syntax

hHandle_t get

Returns

The id of the handle.

Description

Returns the id of the handle_t object.

16.6.10.4 get Method

Syntax

hUuid get

Returns

Returns the value of the UUID object.

16.6.10.5 set Method

Syntax

hHandle_t set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the handle_t object represented by ***hHandle_t***.

16.6.10.6 set Method

Syntax

hUuid_t set *newValue*

Returns

Empty string

Description

Sets the value of the ZUUID object to newValue. If new Value is “” or “NULL”, sets the value of the object to the NULL UUID value.

Example

```
> set u [make- uuid]
> u get
=> 8a781e06-f3d6-11d0-83e6-80606abaaa77
> u set NULL
> u get
```

16.6.10.7 bind Method

Syntax

hHandle_t bind *CDSentry*

Returns

Empty string

Description

Binds the handle_t object to the server represented by ***CDSentry***. ***CDSentry*** is either a server or a group entry in the Cell Directory Service (CDS).

Example

```
> set handle [make-handle_t]
> $handle bind /.:/test/locnet/adv-group
```

Exceptions

CDS entry does not exist

16.6.10.8 setAuthentication Method

Syntax

```
hHandle_t setAuthentication protection authenticationService \
                            authorizationService
```

Returns

Empty string

Description

Sets the authentication of the handle_t object.

The **bind** member function must be called prior to calling this method.

protection is the protection level, which can be one of **default**, **none**, **connect**, **call**, **pkt**, **pkt_integ**, or **pkt_privacy**. The default value is **default**.

authenticationService is the authentication service code, which can be one of **none**, **dce_secret**, **dce_public**, and **default**. The default value is **default**.

authorizationService is the authorization service code, which can be one of **none**, **name**, or **dce**. The default value is **none**.

Example

```
> set handle [make-handle_t]
> $handle bind /.:/test/locnet/adv-group
> $handle setAuthentication pkt_privacy dce_public dce
```

Exceptions

Bind has not been called

16.6.11 hyper

A hyper is an 64-bit signed integer. It can contain the values from -2^{63} to $2^{63}-1$. Since Tcl does not support 64-bit integers, the Tcl interface uses two 32-bit hexadecimal integers to represent the hyper value. The pair of integers is in high-word, low-word format.

16.6.11.1 make-hyper Constructor

Syntax

`make-hyper` *`?initial-value?`*

Returns

A handle to the newly created hyper object.

Description

Creates a hyper object and returns a handle to the hyper.

Exceptions

- Out of memory
- Illegal initial value

16.6.11.2 get Method

Syntax

`hHyper` `get`

Returns

The pair of hexadecimal numbers representing the hyper integer.

Description

Returns the stored number.

16.6.11.3 set Method

Syntax

hHyper set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the hyper object represented by ***hHyper***. ***newValue*** must be a pair of hexadecimal integers.

Example

```
> set i [make-hyper]
> $i set {0000 0005}
> $i get
0000 0005
```

Exceptions

Illegal value for number

16.6.12 long

A long is an 32-bit signed integer. It can contain the values from -2^{31} to $2^{31}-1$.

16.6.12.1 make-long Constructor

Syntax

`make-long` *`?initial-value?`*

Returns

A handle to the newly created long.

Description

Creates a long object and returns a handle to the long.

Exceptions

- Out of memory
- Illegal initial value

16.6.12.2 get Method

Syntax

`hLong` `get`

Returns

A number between -2^{31} to $2^{31}-1$.

Description

Returns thestored number.

16.6.12.3 set Method

Syntax

hLong set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the long object represented by ***hLong***.

Example

```
> set i [make-long]
> $i set 5
> $i get
5
```

Exceptions

Illegal value for number

16.6.13 pipe

A **pipe** is type that allows a large volume of data to be transferred from the client to the server or vice versa. This describes the interface to a generic enumerated type.

This implementation of pipes uses files as the source and sink for the transfer of pipe data. There are two different sets of operations based on whether the pipe is used in an emulated client or an emulated server. Note also that there are two methods for each side of the transfer: one for an input pipe and another for an output pipe. If a pipe is bidirectional (ie, transfers both input and output data), both methods are available.

16.6.13.1 make-*pipe* Constructor

Syntax

`make-`*`pipe`*

Returns

A handle to the newly created pipe object.

Description

Creates a pipe object and returns a handle to the object.

Exceptions

Out of memory

16.6.13.2 setInputFilename Method

Syntax

`hPipe` `setInputFilename` *`filename`*

Returns

Empty string

Description

The **setInputFilename** method sets the source for an input pipe. It is only available to the client emulator. This method is a setup step. The file is not opened until an RPC that uses the pipe is called.

If the file can not be created, a DCE runtime exception will be thrown.

16.6.13.3 setOutputFilename Method

Syntax

hPipe `setOutputFilename` *`filename`*

Returns

Empty string

Description

The **setOutputFilename** method sets the sink for an output pipe. It is only available to the client emulator. This method is a setup step. The file is not opened until an RPC that uses the pipe is called.

If the file can not be opened, a DCE runtime exception will be thrown.

16.6.13.4 dumpFile Method

Syntax

hPipe `dumpFile` *`filename`*

Returns

Empty string

Description

The **dumpFile** method reads the data from the input pipe and dumps it to the file. It is used by the emulated server.

Unlike the client methods, this method reads the data from the pipe and stores it in the file immediately.

Exceptions

Output file, ***filename***, can not be created

16.6.13.5 readFile Method

Syntax

hPipe readFile *filename*

Returns

Empty string

Description

The **readFile** method reads the data from the file and sends it to the output pipe. It is used by the emulated server.

Unlike the client methods, this method opens the file immediately.

Exceptions

Input file, ***filename***, does not exist

16.6.14 pointer

A pointer is an indirect reference to another object. Whereas in **C**, a pointer has a specific type, these pointers are generic. There is no type checking performed on the values being stored in a pointer. The string **NULL** is used to denote a NULL pointer.

16.6.14.1 make-pointer Constructor

Syntax

```
make-pointer ?initial-value?
```

Returns

A handle to the newly created pointer object.

Description

Creates a pointer object and returns a handle to the pointer. If ***initialValue*** is specified, it is the handle to another object — the object towhich the pointer is pointing.

Exceptions

- Out of memory
- Illegal initial value

16.6.14.2 get Method

Syntax

```
hPointer get
```

Returns

The handle of the object to which the pointer is pointing.

Description

Returns the target object or **NULL** if the object is not pointing to another object.

16.6.14.3 set Method

Syntax

hPointer set *newValue*

Returns

Empty string

Description

Makes the pointer point to the object associated with ***newValue***.

Exceptions

- Pointer points to **NULL**
- Illegal value

16.6.14.4 -> (dereference) Method

Syntax

hPointer -> method args

Returns

The result of invoking ***method*** on the object being pointed to.

Description

Dereferences the pointer object and invokes the specified method (***method***) on the dereferenced object with the associated arguments (***args***).

Example

```
> set p [make-pointer]
> set f [make-float 3.14159265]
> $p set $f
> $p -> get
3.14159265
> $p -> set 0.0
> $f get
0.0
```

Exceptions

- Pointer points to **NULL**
- ***method*** and/or ***args*** does not apply to dereferenced object

16.6.14.5 get-pointer-contents Method

Syntax

hPointer `get-pointer-contents`

Returns

The address stored in the real pointer.

Description

Returns the address that is stored in the real pointer, not the name of the handle associated with that object. This method is provided for debugging purposes.

16.6.15 short

A short is an 16-bit signed integer. It can contain the values from -32768 to 32767.

16.6.15.1 make-short Constructor

Syntax

```
make-short ?initial-value?
```

Returns

A handle to the newly created short object.

Description

Creates a short object and returns a handle to the short.

Exceptions

- Out of memory
- Illegal initial value

16.6.15.2 get Method

Syntax

```
hShort get
```

Returns

A number between -32768 and 32767.

Description

Returns the stored number.

16.6.15.3 set Method

Syntax

hShort set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the short object represented by ***hShort***.

Example

```
> set i [make-short]
> $i set 5
> $i get
5
```

Exceptions

Illegal value for number

16.6.16 small

A small is an 8-bit signed integer. It can contain the values from -128 to 127.

16.6.16.1 make-small Constructor

Syntax

```
make-small ?initial-value?
```

Returns

A handle to the newly created small object.

Description

Creates a small object and returns a handle to the small.

Exceptions

- Out of memory
- Illegal initial value

16.6.16.2 get Method

Syntax

```
hSmall get
```

Returns

A number between -128 and 127.

Description

Returns the stored number.

16.6.16.3 set Method

Syntax

hSmall set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the small object represented by ***hSmall***.

Example

```
> set i [make-small]
> $i set 5
> $i get
5
```

Exceptions

Illegal value for number

16.6.17 string

A string is an null-terminated ASCII character sequence.

16.6.17.1 make-string Constructor

Syntax

```
make-string ?initial-value?
```

Returns

A handle to the newly created string object.

Description

Creates a string object and returns a handle to the string.

Exceptions

Out of memory

16.6.17.2 get Method

Syntax

```
hString get
```

Returns

A string.

Description

Returns the stored string.

16.6.17.3 set Method

Syntax

hString set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the string object represented by ***hString***.

Example

```
> set s [make-string]
> $s set {abcdefg hijklmnop}
> $s get
abcdefg kijklmnop
```

16.6.18 structure

A structure is a user-defined type that is a group of related data objects. This describes the interface to a generic structure.

There are two constructors available: one for structures that contain a conformant array and one for structures without a conformant array. A conformant array is an array with a size determined at runtime. The second constructor is provided to set this runtime size.

16.6.18.1 make-*struct* Constructor

Syntax

`make-`*`struct`*

Returns

A handle to the newly created structure object.

Description

Creates an structure object and returns a handle to the object.

Exceptions

Out of memory

16.6.18.2 make-*struct* Constructor Containing a Conformant Array

Syntax

`make-`*`struct size`*

Returns

A handle to the newly created structure.

Description

Creates an structure object and returns a handle to the object. This is used for structures that contain a conformant array. The size parameter determines the number of elements in the conformant array.

Exceptions

Out of memory

16.6.18.3 members Method

Syntax

hStruct members

Returns

The names of all members of the structure.

Description

Returns a list of the names of the members of the structure.

16.6.18.4 *memberName* Method

Syntax

hStruct memberName ?method args?

Returns

The result of applying ***method*** to the member given by ***memberName***.

Description

Retrieves the member of the structure using ***memberName***. Invokes ***method*** and ***args*** on this member. If ***method*** is not given, this returns the name of the member object.

Example

```
> set envTrace [make-xbbHdr_t]
> $envTrace trace set idl_TRACE_A
> $envTrace log set idl_LOG_LVL2
> $envTrace trace get
idl_TRACE_A
> $envTrace log get
idl_LOG_LVL2
```

Exceptions

- No member with name ***memberName***
- Can’t apply ***method*** and/or ***args*** to member object

16.6.19 uhyper

A uhyper is an 64-bit unsigned integer. It can contain the values from 0 to 2^{64}-1. Like its signed counterpart, access to the integer is achieved through a pair of high and low words.

16.6.19.1 make-uhyper Constructor

Syntax

`make-uhyper` *`?initial-value?`*

Returns

A handle to the newly created uhyper object.

Description

Creates au hyper object and returns a handle to the uhyper.

Exceptions

- Out of memory
- Illegal initial value

16.6.19.2 get Method

Syntax

`hUhyper` `get`

Returns

The pair of hexadecimal numbers representing the uhyper integer.

Description

Returns the stored number.

16.6.19.3 set Method

Syntax

hUhyper set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the uhyper object represented by ***hUhyper***. ***newValue*** must be a pair of hexadecimal integers.

Example

```
> set i [make-uhyper]
> $i set {0000 0005}
> $i get
0000 0005
```

Exceptions

Illegal value for number

16.6.20 ulong

A ulong is an 32-bit unsigned integer. It can contain the values from 0 to 2^{32}-1.

16.6.20.1 make-ulong Constructor

Syntax

`make-ulong` *`?initial-value?`*

Returns

A handle to the newly created ulong object.

Description

Creates a long object and returns a handle to the ulong.

Exceptions

- Out of memory
- Illegal initial value

16.6.20.2 get Method

Syntax

`hUlong` `get`

Returns

A number between 0 and 2^{32}-1.

Description

Returns the stored number.

16.6.20.3 set Method

Syntax

hUlong set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the ulong object represented by ***hUlong***.

Example

```
> set i [make-ulong]
> $i set 5
> $i get
5
```

Exceptions

Illegal value for number

16.6.21 union

A union is a user-defined type, much like a structure. However, all members shared the same memory, so only one member is valid at any given time. A union has an additional member, known as the discriminant, which is valid at all times. The discriminant is the key that determines which of the other members is valid. In order to access a member, that member must first be valid.

This describes the interface to a generic union.

16.6.21.1 make-*union* Constructor

Syntax

`make-`*`union`*

Returns

A handle to the newly created union object.

Description

Creates a union object and returns a handle to the object.

Exceptions

Out of memory

16.6.21.2 members Method

Syntax

`hUnion` `members`

Returns

The names of all members of the union.

Description

Returns a list of the names of the members of the union. This list does not include the discriminant.

16.6.21.3 *memberName* Method

Syntax

```
hUnion memberName ?method args?
```

Returns

The result of applying ***method*** to the member given by ***memberName***.

Description

Retrieves the member of the union using ***memberName***. Invokes ***method*** and ***args*** on this member. If ***method*** is not given, returns the name of the member object.

If the member is not currently valid, an exception is thrown.

Exceptions

- No member with name ***memberName***
- Can't apply ***method*** and/or ***args*** to member object
- Member is not valid

16.6.21.4 tagName Method

Syntax

```
hUnion tagName
```

Returns

The name of the tag containing the discriminant of the union.

Description

Returns the name of the discriminant of the union.

16.6.21.5 *tagName* Method to Retrieve Discriminant

Syntax

```
hUnion tagName ?method args?
```

Returns

The result of applying ***method*** to the discriminant given by ***tagName***.

Description

Retrieves the discriminant of the union using ***tagName***. Invokes ***method*** and ***args*** on this member. If ***method*** is not given, returns the name of the discriminant object.

Exceptions

- No member with name ***memberName***
- Can't apply ***method*** and/or ***args*** to discriminant object

16.6.21.6 currentTag Method

Syntax

```
hUnion currentTag
```

Returns

The name of the union member that is currently valid (based on discriminant).

Description

Returns the name of the union member that is currently valid. This depends on the value of the discriminant of the union.

Example

```
> set u [make-myUnion]
> $u tagName
theTag
> $u currentTag
alpha
> $u alpha get
a
> $u beta get
error: beta is not active; alpha is currently active
```

Exceptions

Discriminant value out of domain

16.6.22 ushort

A ushort is an 16-bit unsigned integer. It can contain the values from 0 to 65535.

16.6.22.1 make-ushort Constructor

Syntax

```
make-ushort ?initial-value?
```

Returns

A handle to the newly created ushort object.

Description

Creates a short object and returns a handle to the ushort.

Exceptions

- Out of memory
- Illegal initial value

16.6.22.2 get

Syntax

```
hUshort get
```

Returns

A number between 0 and 65535.

Description

Returns the stored number.

16.6.22.3 set

Syntax

hUshort set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the ushort object represented by ***hUshort***.

Example

```
> set i [make-ushort]
> $i set 5
> $i get
5
```

Exceptions

Illegal value for number

16.6.23 usmall

A **usmall** is an 8-bit unsigned integer. It can contain the values from 0 to 255.

16.6.23.1 make-usmall Constructor

Syntax

```
make-usmall ?initial-value?
```

Returns

A handle to the newly created usmall object.

Description

Creates a small object and returns a handle to the usmall.

Exceptions

- Out of memory
- Illegal initial value

16.6.23.2 get Method

Syntax

```
hUsmall get
```

Returns

A number between 0 and 255.

Description

Returns the stored number.

16.6.23.3 set Method

Syntax

hUsmall set *newValue*

Returns

Empty string

Description

Stores ***newValue*** into the usmall object represented by ***hUsmall***.

Example

```
> set i [make-usmall]
> $i set 5
> $i get
5
```

Exceptions

Illegal value for number

16.7 RPC Calls in the Emulated Client

ATcl command is created for each operation defined in the interface. The command name is the name of the operation. The parameters to the command are the handles to the actual data objects to be used for the transfer. All handles must point to valid data objects. Input parameters must be initialized; output parameters do not have to be initialized.

Syntax

RPC ?arg ...?

Returns

Handle to the return value of the RPC call, as defined in the interface definition.

Description

This is how to call an RPC. Each parameter in the interface definition (including output parameters) must be provided via handles to the real data.

Example

In this example, result is an **error_status_t** object which contains the DCE status of the RPC call. The value **rpc_s_okay** means that the call was successful from DCE's point of view.

```
> set result [xbdValidateAddr $handle $streetAddr \
                 $status $msg $output]
> $result get
rpc_s_okay
```

Exceptions

- Call timed out
- ***RPC*** is not defined in the interface
- Wrong number of arguments

16.8 Printing Objects

16.8.1 print

The following print routine is valid for all types defined in the IDL.

Syntax

$handle print *label ?fileId?*

Returns

Printout of the specified object or an empty string.

Description

The **print** method dumps the contents of a data handle in text format. If the object contains nested objects (e.g., arrays, structures, unions, pointers), the contained objects are also displayed.

If ***fileId*** is provided, the output goes to the file handle. ***fileId*** must be created using Tcl's open command. If ***fileId*** is not provided, the output goes to standard output.

label is used as the label of the main object.

Example

```
> set x [make-mystruct1_t]
> $x print x
   x (struct) = 'mystruct1_t' {
      a_char (char) = 'c'
      a_short (short) = 0
   }
```

Exceptions

fileId was not opened for writing

16.9 Getting the Type of an Object - typeOfHandle

All objects may retrieve their type using the **typeOfHandle** method.

Syntax

handle typeOfHandle

Returns

A string representing the type of the object

Description

The **typeOfHandle** method returns the type of the object. The current types returned are **BYTE**, **CHAR**, **SMALL**, **USMALL**, **SHORT**, **USHORT**, **LONG**, **ULONG**, **HYPER**, **UHYPER**, **FLOAT**, **DOUBLE**, **BUFFER**, **STRING**, **ENUM**, **STRUCT**, **UNION**, **ARRAY**, **POINTER**, **HANDLE_T**, **ERROR_STATUS_T**, **BOOLEAN**, **PIPE**.

Example

```
> set x [make-mystruct1_t]

> $x typeOfHandle
STRUCT
```

16.10 RPC Calls in the Emulated Server

In the emulated server, DCE is in control until an RPC request is made. At that point, the Tcl command with the name of the operation is called. Handles are created for each of the parameters; these handles are passed on the command line. The Tcl command is a user-defined procedure which is responsible for producing valid output values in the output parameters, and for returning a valid return value.

The following procedure represents a template for the RPC procedures. In this example, the operation takes two input parameters (***handle*** and ***streetAddr***), and produces three output parameters (***status***, ***msg***, and ***output***). In addition, the operation returns an object of type **error_status_t** indicating the status of the RPC call.

```
proc xbdValidateAddr {handle streetAddr status msg output} {
     puts "entered xbdValidateAddr"
     # display streetAddr
     ...
     # compute status, msg, and output
     ...
     return [make-error_status_t rpc_s_okay]
}
```

If any error occurs within a procedure for an operation, the emulated server replies with an error status of **rpc_s_unknown_reject**.

16.11 Constants

A Tcl command is created for each constant defined in the interface. The command name is the name of the constant. The return value of the command is the value of the constant, as a string.

Syntax

constantName

Returns

The value of the constant defined in the interface

Description

Constants can represent numeric or textural values.

16.12 Destroying Objects

All objects instantiated may be removed using the **destroy** method.

16.12.1 destroy

Syntax

handle destroy

Returns

An empty string

Description

The **destroy** method removes the handle from Tcl's command space. It also frees any resources associated with the handle. Subsequent use of the handle will result in an error.

If the object is part of a larger object (e.g., a member of a structure), the memory associated with the object is not freed until the container object is deleted. Only the name of the Tcl handle is deleted.

This is equivalent to using the low-level Tcl **rename** command as follows in the example.

Example

Both of these are equivalent:

```
> $object destroy
> rename $object {}
```

16.13 Deleting Handles and Objects

The **xmyDceScope** method is used to automatically delete objects that were created. In addition, there are a series of methods that can be used to support **xmyDceScope**.

16.13.1 xmyDceScope

Syntax

`xmyDceScope` *`body`*

Returns

The result of executing ***body***

Description

The **xmyDceScope** command evaluates the Tcl commands in ***body***, ensuring that all objects created in ***body*** are deleted when finished. This is done regardless of the reason that ***body*** exits, through error, normal exit, or one of Tcl's flow control statments (**return**, **continue**, **break**).

This method is particularly useful around calls to RPCs to ensure that all data produced as a result of calling the RPC is deleted.

There is currently a limitation that there can only be one scope active at a time. Therefore, scopes can not be nested.

Example

```
> xmyDceScope {
   set in [make-long 5];       # create an input parameter
   set out [make-pointer];     # create an output parameter
   rpc-call $in $out;          # call the RPC
   $out print out;             # print the output parameter
};                            # delete in and out
```

Exceptions

Another scope is active

16.13.2 Methods Supporting the Deletion of Objects

These commands are provided for backward-compatibility. They should not be used in new scripts. The new **xmyDceScope** command should be used instead.

16.13.2.1 xmyDceDeleteHandles

Syntax

```
xmyDceDeleteHandles ?handle1? ... ?HandleN?
```

Returns

An empty string

Description

The **xmyDceDeleteHandles** command deletes all the handles given as arguments. There is no error checking done to determine if the handles exist. This is used for bulk deletion of handles.

Examples

This deletes the handles ***clientConn1*** and ***clientConn2***.

```
> xmyDceDeleteHandles clientConn1 clientConn2
```

This delete all handles.

```
> xmyDceDeleteHandles [info commands .xmyDceBinding*]
```

16.13.2.2 xmyDceDeleteAllHandles

Syntax

```
xmyDceDeleteAllHandles
```

Returns

An empty string

Description

The **xmyDceDeleteAllHandles** command deletes all handles used by DCE. These are commands that match **.xmyDceBinding***. There is no error checking.

Examples

These two example are equivalent.

```
> xmyDceDeleteHandles [info commands .xmyDceBinding*]
> xmyDceDeleteAllHandles
```

16.13.2.3 xmyDceDeleteDataHandles

Syntax

```
xmyDceDeleteDataHandles
```

Returns

An empty string

Description

The **xmyDceDeleteDataHandles** command deletes all handles that are not of type **handle_t**.

Example

This delete all data handles.

```
> xmyDceDeleteDataHandles
```

16.13.2.4 xmyDceSaveHandles

Syntax

DCE Package:Handles:Restoring

```
xmyDceSaveHandles
```

Returns

An empty string

Description

The **xmyDceSaveHandles** command saves the list of all handles currently in use. The **xmyDceRestoreHandles** command deletes all handles not in this list. The list is stored in the **xmyG_HandlesToSave** associative array.

Example

This saves and restores a set of handles.

```
> xmyDceSaveHandles
> #contents of a script
> xmyDceRestoreHandles
```

16.13.2.5 xmyDceRestoreHandles

Syntax

```
xmyDceRestoreHandles
```

Returns

An empty string

Description

The **xmyDceRestoreHandles** command restores the set of handles to the set previously marked with **xmyDceSaveHandles**. It does this by deleting all of the handles that were not marked using **xmyDceSaveHandles**.

Example

This saves and restores a set of handles.

```
> xmyDceSaveHandles
> # contents of a script
> xmyDceRestoreHandles
```

16.14 Getting the Interface- xmyDceInterface

You may get the name of the current interface handle using the **xmyDceInterface** command.

Syntax

```
xmyDceInterface
```

Returns

The name of the interface handle

Description

The **xmyDceInterface** command returns the name of the interface that is being used in the emulated client or emulated server.

Example

```
> set interface [xmyDceInterface]
> $interface name
tutorial
> $interface uuid
000b6ed6-debb-11a1-bcc2-80606abaaa77
```

16.15 DCE/Async Commands

The **xmyDceRecordEnterOperation**, **xmyDceRecordExitOperation**, and **xmyDceCallRpc** commands are used for integrating with the MYNAH System using the TermAsync package. They are only used in the emulated client, and have no effect if not running under the TermAsync package (i.e., **XMY_DCE_RUNNING_UNDER_ASYNC** is not set).

These commands are used in the template scripts produced by the parser.

16.15.1 xmyDceRecordEnterOperation

Syntax

`xmyDceRecordEnterOperation` *`operationName`*

Returns

An empty string

Description

The **xmyDceRecordEnterOperation** command records the entry into the operation.

Example

```
> xmyDceRecordEnterOperation rpc-1
> rpc-1 $handle $output;           # call the RPC
> $output print output;            # dump the output parameter
> xmyDceRecordExitOperation rpc-1 “OKAY”
```

16.15.2 xmyDceRecordExitOperation

Syntax

```
xmyDceRecordExitOperation operationName status
```

Returns

An empty string

Description

The **xmyDceRecordExitOperation** command records the exit status of the operation.

status must be one of **OKAY**, **EVAL_FAILED**, **PARAM_BAD**, **ILLEGAL_RETURN**, **RETURN_BAD**, **CLIENT_OP_FAILED**

Example

```
xmyDceRecordEnterOperation rpc-1
rpc-1 $handle $output;          # call the RPC
$output print output;           # dump the output parameter
xmyDceRecordExitOperation rpc-1 “OKAY”
```

16.15.3 xmyDceCallRpc

Syntax

```
xmyDceCallRpc rpc ?args?
```

Returns

The result of calling ***rpc***

Description

The **xmyDceCallRpc** command calls the RPC with the arguments, catching any Tcl error response. If an error occurs, it calls **xmyDceRecordExitOperation** with **CLIENT_OP_FAILED** status and rethrows the error.

Example

Record an exit due to an error return from the RPC.

```
xmyDceRecordEnterOperation rpc-1
xmyDceCallRpc rpc-1 $handle $output; # call the RPC
$output print output;                # dump the outputs
xmyDceRecordExitOperation rpc-1 “OKAY”
```

17. GUI Tcl Language Extensions

The GUI test domain of the MYNAH system provides the ability to test GUI applications running under the X Window System or Microsoft® Windows™ 3.1, Windows95, or Windows NT® systems. It does this by interfacing to existing vendor tools (e.g., QA Partner™, QC/Replay™, and XRunner®).

There are no language extensions provided specifically for the GUI test domain. Users write test scripts for the specific tool they are using. The MYNAH System treats these scripts just like any other MYNAH script. They must, however, be executed as special engines that are configured for the appropriate vendor toolThey may execute the scripts using the MYNAH GUI or the MYNAH CLUI, or as a child script of another MYNAH script using the Child Script language extensions. See the description of Child Script language extensions in Section 7.

Appendix A: Basic Tcl Commands

This Appendix contains hard copy versions of the manual pages for the the basic Tcl commands.

As in Section 5, in this Appendix the terms *fileName* and *fileId* are used. While they seem similar, possibly synonymous, they are very different.

fileName is the name of a file in the operating system, and ***fileId*** is an identifier Tcl creates when you open ***fileName*** using the **open** (Section A.39) command; ***fileId*** is a handle to the open ***fileName***. For example, assume you have a file called *Login_script1* and you want to open it. You could type the following:

```
> open Login_script1
file3
```

Login_script1 is a ***fileName*** and **file3** is a ***fileId***. As with all handles, you can assign a ***fileId*** to a variable, such as in

```
> set x [open Login_script1]
file3
```

A.1 append

Syntax

append *varName value ?value value ...?*

Description

The **append** command append all of the *value* arguments to the current value of variable **varName**. If **varName** doesn't exist, it is given a value equal to the concatenation of all the ***value*** arguments. This command provides an efficient way to build up long variables incrementally. For example

```
append a $b
```

is much more efficient than

```
set a $a$b
```

if **$a** is long.

A.2 array

Syntax

```
array option arrayName ?arg arg ...?
```

Description

The **array** command performs one of several operations on the variable given by ***arrayName***. ***arrayname*** must be the name of an existing array variable. The **option** argument determines what action is carried out by the command. The legal **options** (which may be abbreviated) are:

array anymore *arrayName searchId*
: Returns 1 if there are any more elements left to be processed in an array search, 0 if all elements have already been returned. *SearchId* indicates which search on *arrayName* to check, and must have been the return value from a previous invocation of **array startsearch**. This option is particularly useful if an array has an element with an empty name, since the return value from **array nextelement** won't indicate whether the search has been completed.

array donesearch *arrayName searchId*
: This command terminates an array search and destroys all the state associated with that search. *SearchId* indicates which search on *arrayName* to destroy, and must have been the return value from a previous invocation of **array startsearch**. Returns an empty string.

array names *arrayName*
: Returns a list containing the names of all of the elements in the array. If there are no elements in the array then an empty string is returned.

array nextelement *arrayName searchId*
: Returns the name of the next element in *arrayName*, or an empty string if all elements of *arrayName* have already been returned in this search. The *searchId* argument identifies the search, and must have been the return value of an **array startsearch** command. Warning: if elements are added to or deleted from the array, then all searches are automatically terminated just as if **array donesearch** had been invoked; this will cause **array nextelement** operations to fail for those searches.

array size *arrayName*
: Returns a decimal string giving the number of elements in the array.

array startsearch *arrayName*

This command initializes an element-by-element search through the array given by *arrayName*, such that invocations of the **array nextelement** command will return the names of the individual elements in the array. When the search has been completed, the **array donesearch** command should be invoked. The return value is a search identifier that must be used in **array nextelement** and **array donesearch** commands; it allows multiple searches to be underway simultaneously for the same array.

A.3 break

Syntax

```
break
```

Description

The **break** command may be invoked only inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL_BREAK code to signal the innermost containing loop command to return immediately.

A.4 case

Syntax

```
case string ?in? patList body ?patList body ...?
case string ?in? {patList body ?patList body ...?}
```

Description

> **NOTE —** The **case** command is obsolete and is supported only for backward compatibility. At some point in the future it may be removed entirely. You should use the **switch** command instead.

The **case** command matches *string* against each of the *patList* arguments in order. Each *patList* argument is a list of one or more patterns. If any of these patterns matches *string* then **case** evaluates the following *body* argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. Each *patList* argument consists of a single pattern or list of patterns. Each pattern may contain any of the wild-cards described under **string match**. If a *patList* argument is **default**, the corresponding body will be evaluated if no *patList* matches *string*. If no *patList* argument matches *string* and no default is given, then the **case** command returns an empty string.

Two syntaxes are provided for the *patList* and *body* arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line case commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *patList* arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

A.5 catch

Syntax

```
catch script ?varName?
```

Description

The **catch** command may be used to prevent errors from aborting command interpretation. **catch** calls the Tcl interpreter recursively to execute ***script***, and always returns a TCL_OK code, regardless of any errors that might occur while executing *script*. The return value from **catch** is a decimal string giving the code returned by the Tcl interpreter after executing *script*. This will be **0** (TCL_OK) if there were no errors in *script*; otherwise it will have a non-zero value corresponding to one of the exceptional return codes (see tcl.h for the definitions of code values). If the *varName* argument is given, then it gives the name of a variable; **catch** will set the variable to the string returned from *script* (either a result or an error message).

A.6 cd

Syntax

```
cd ?dirName?
```

Description

cd change the current working directory to ***dirName***, or to the home directory (as specified in the HOME environment variable) if ***dirName*** is not given. If ***dirName*** starts with a tilde, then tilde-expansion is done as described for **Tcl_TildeSubst**. Returns an empty string.

A.7 close

Syntax

```
close fileId
```

Description

Closes the file given by *fileId*. *fileid* must be the return value from a previous invocation of the **open** command; after this command, it should not be used anymore. If *fileId* refers to a command pipeline instead of a file, then **close** waits for the children to complete. The normal result of this command is an empty string, but errors are returned if there are problems in closing the file or waiting for children to complete.

A.8 concat

Syntax

```
concat ?arg arg ...?
```

Description

This command treats each argument as a list and concatenates them into a single list. It also eliminates leading and trailing spaces in the *arg*'s and adds a single separator space between *arg*'s. It permits any number of arguments. For example, the command

```
concat a b {c d e} {f {g h}}
```

will return

```
a b c d e f {g h}
```

as its result.

If no *arg*s are supplied, the result is an empty string.

A.9 continue

Syntax

```
continue
```

Description

The **continue** command may be invoked only inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL_CONTINUE code to signal the innermost containing loop command to skip the remainder of the loop's body but continue with the next iteration of the loop.

A.10 eof

Syntax

```
eof fileId
```

Description

The **eof** command returns 1 if an end-of-file condition has occurred on *fileId*, 0 otherwise. *fileid* must have been the return value from a previous call to **open**, or it may be **stdin**, **stdout**, or **stderr** to refer to one of the standard I/O channels.

A.11 error

Syntax

`error` *`message ?info? ?code?`*

Description

Returns a TCL_ERROR code, which causes command interpretation to be unwound. *Message* is a string that is returned to the application to indicate what went wrong.

If the *info* argument is provided and is non-empty, it is used to initialize the global variable **errorInfo**. **errorInfo** is used to accumulate a stack trace of what was in progress when an error occurred; as nested commands unwind, the Tcl interpreter adds information to **errorInfo**. If the *info* argument is present, it is used to initialize **errorInfo** and the first increment of unwind information will not be added by the Tcl interpreter. In other words, the command containing the **error** command will not appear in **errorInfo**; in its place will be *info*. This feature is most useful in conjunction with the **catch** command: if a caught error cannot be handled successfully, *info* can be used to return a stack trace reflecting the original point of occurrence of the error:

```
catch {...} errMsg set savedInfo $errorInfo ... error \
$errMsg $savedInfo
```

If the *code* argument is present, then its value is stored in the **errorCode** global variable. This variable is intended to hold a machine-readable description of the error in cases where such information is available; see the section BUILT-IN VARIABLES below for information on the proper format for the variable. If the *code* argument is not present, then **errorCode** is automatically reset to ‘‘NONE’’ by the Tcl interpreter as part of processing the error generated by the command.

A.12 eval

Syntax

```
eval arg ?arg ...?
```

Description

The **eval** command concatenates all its arguments in the same fashion as the **concat** command, passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it). **eval** takes one or more arguments, which together comprise a Tcl script containing one or more commands.

A.13 exec

Syntax

exec *?switches? arg ?arg ...?*

Description

The **exec** command treats its arguments as the specification of one or more subprocesses to execute. The arguments take the form of a standard shell pipeline where each *arg* becomes one word of a command, and each distinct command becomes a subprocess.

If the initial arguments to **exec** start with **–** then they are treated as command-line switches and are not part of the pipeline specification. The following switches are currently supported:

–keepnewline Retains a trailing newline in the pipeline's output. Normally a trailing newline will be deleted.

– – Marks the end of switches. The argument following this one will be treated as the first arg even if it starts with a –.

If an *arg* (or pair of *arg*'s) has one of the forms described below then it is used by **exec** to control the flow of input and output among the subprocess(es). Such arguments will not be passed to the subprocess(es). In forms such as ''< *fileName*'' *fileName* may either be in a separate argument from ''<'' or in the same argument with no intervening space (i.e. ''<*fileName*'').

| Separates distinct commands in the pipeline. The standard output of the preceding command will be piped into the standard input of the next command.

|& Separates distinct commands in the pipeline. Both standard output and standard error of the preceding command will be piped into the standard input of the next command. This form of redirection overrides forms such as 2> and >&.

< *fileName* The file named by ***fileName*** is opened and used as the standard input for the first command in the pipeline.

<@ *fileId* ***fileid*** must be the identifier for an open file, such as the return value from a previous call to **open**. It is used as the standard input for the first command in the pipeline. *FileId* must have been opened for reading.

<< *value* *Value* is passed to the first command as its standard input.

> *fileName* Standard output from the last command is redirected to the file named *fileName*, overwriting its previous contents.

2> *fileName*	Standard error from all commands in the pipeline is redirected to the file named *fileName*, overwriting its previous contents.
>& *fileName*	Both standard output from the last command and standard error from all commands are redirected to the file named *fileName*, overwriting its previous contents.
>> *fileName*	Standard output from the last command is redirected to the file named *fileName*, appending to it rather than overwriting it.
2>> *fileName*	Standard error from all commands in the pipeline is redirected to the file named *fileName*, appending to it rather than overwriting it.
>>& *fileName*	Both standard output from the last command and standard error from all commands are redirected to the file named *fileName*, appending to it rather than overwriting it.
>@ *fileId*	***fileid*** must be the identifier for an open file, such as the return value from a previous call to **open**. Standard output from the last command is redirected to *fileId*'s file, which must have been opened for writing.
2>@ *fileId*	***fileid*** must be the identifier for an open file, such as the return value from a previous call to **open**. Standard error from all commands in the pipeline is redirected to *fileId*'s file. The file must have been opened for writing.
>&@ *fileId*	***fileid*** must be the identifier for an open file, such as the return value from a previous call to **open**. Both standard output from the last command and standard error from all commands are redirected to *fileId*'s file. The file must have been opened for writing.

If standard output has not been redirected then the **exec** command returns the standard output from the last command in the pipeline. If any of the commands in the pipeline exit abnormally or are killed or suspended, then **exec** will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the **errorCode** variable will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file and that standard error isn't redirected, then **exec** will return an error; the error message will include the pipeline's standard output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is normally deleted from the result or error message. This is consistent with other Tcl return values, which don't normally end with newlines.

However, if **–keepnewline** is specified then the trailing newline is retained.

If standard input isn't redirected with ''<'' or ''<<'' or ''<@'' then the standard input for the first command in the pipeline is taken from the application's current standard input.

If the last *arg* is ''&'' then the pipeline will be executed in background. In this case the **exec** command will return a list whose elements are the process identifiers for all of the subprocesses in the pipeline.

The standard output from the last command in the pipeline will go to the application's standard output if it hasn't been redirected, and error output from all of the commands in the pipeline will go to the application's standard error file unless redirected.

The first word in each command is taken as the command name; tilde-substitution is performed on it, and if the result contains no slashes then the directories in the PATH environment variable are searched for an executable by the given name. If the name contains a slash then it must refer to an executable reachable from the current directory. No ''glob'' expansion or other shell-like substitutions are performed on the arguments to commands.

A.14 exit

Syntax

```
exit ?returnCode?
```

Description

Terminate the process, returning *returnCode* to the system as the exit status. If *returnCode* isn't specified then it defaults to 0.

A.15 expr

Syntax

```
expr arg ?arg arg ...?
```

Description

Concatenates *arg*'s (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value.

The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

```
expr 8.2 + 6
```

evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons.

Operands

A Tcl expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression processor. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is **0**), or in hexadecimal (if the first two characters of the operand are **0x**). If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the ''f'', ''F'', ''l'', and ''L'' suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

[1] As an numeric value, either integer or floating-point.

[2] As a Tcl variable, using standard **$** notation. The variable's value will be used as the operand.

[3] As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand

[4] As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.

[5] As a Tcl command enclosed in brackets. The command will be executed and its result will be used as the operand.

[6] As a mathematical function whose arguments have any of the above forms for operands, such as ‘‘**sin($x)**’’. See below for a list of defined functions.

Where substitutions occur above (e.g. inside quoted strings), they are performed by the expression processor. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents.

For some examples of simple expressions, suppose the variable **a** has the value 3 and the variable **b** has the value 6. Then the command on the left side of each of the lines below will produce the value on the right side of the line:

```
expr 3.1 + $a                 6.1
expr 2 + "$a.$b"              5.6
expr 4* [llength "6 2"]       8
expr {{word one} < "word $a"} 0
```

Operators

The valid operators are listed below, grouped in decreasing order of precedence:

– ~ !	Unary minus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.
*** / %**	Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.
+ –	Add and subtract. Valid for any numeric operands.
<< >>	Left and right shift. Valid for integer operands only.
< > <= >=	Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.
== !=	Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.
&	Bit-wise AND. Valid for integer operands only.
^	Bit-wise exclusive OR. Valid for integer operands only.

|	Bit-wise OR. Valid for integer operands only.
&&	Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for numeric operands only (integers or floating-point).
||	Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for numeric operands only (integers or floating-point).
*x***?***y***:***z*	If-then-else, as in C. If *x* evaluates to non-zero, then the result is the value of *y*. Otherwise the result is the value of *z*. The *x* operand must have a numeric value.

See the C manual for more details on the results produced by each operator. All of the binary operators group left-to-right within the same precedence level. For example, the command

```
expr 4*2 < 7
```

returns 0.

The **&&**, ||, and **?:** operators have ''lazy evaluation'', just as in C, which means that operands are not evaluated if they are not needed to determine the outcome. For example, in the command

```
expr {$v ? [a] : [b]}
```

only one of **[a]** or **[b]** will actually be evaluated, depending on the value of **$v**. Note, however, that this is only true if the entire expression is enclosed in braces; otherwise the Tcl parser will evaluate both **[a]** and **[b]** before invoking the **expr** command.

Math Functions

Tcl supports the following mathematical functions in expressions:

acos cos hypot sinh asincosh log sqrt atanexp log10 tan atan2 floor pow tanh ceilfmod sin

Each of these functions invokes the math library function of the same name; see the manual entries for the library functions for details on what they do. Tcl also implements the following functions for conversion between integers and floating-point numbers:

abs(*arg***)**	Returns the absolute value of arg. Arg may be either integer or floating-point, and the result is returned in the same form.
double(*arg***)**	If *arg* is a floating value, returns *arg*, otherwise converts *arg* to floating and returns the converted value.
int(*arg***)**	If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by truncation and returns the converted value.

round(*arg***)** If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by rounding and returns the converted value.

In addition to these predifined functions, applications may define additional functions using **Tcl_CreateMathFunc**().

Types, Overflow, and Precision

All internal computations involving integers are done with the C type *long*, and all internal computations involving floating-point are done with the C type *double*. When converting a string to floating-point, exponent overflow is detected and results in a Tcl error. For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, integer overflow and underflow are generally not detected reliably for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

```
expr 5 / 4
```

returns 1, while

```
expr 5 / 4.0 expr 5 / ( [string length "abcd"] + 0.0 )
```

both return 1.25.

Floating-point values are always returned with a ‘‘.’’ or an ‘‘e’’ so that they will not look like integer values. For example,

```
expr 20.0/5.0
```

returns ‘‘4.0’’, not ‘‘4’’. The global variable **tcl_precision** determines the the number of significant digits that are retained when floating values are converted to strings (except that trailing zeroes are omitted). If **tcl_precision** is unset then 6 digits of precision are used. To retain all of the significant bits of an IEEE floating-point number set **tcl_precision** to 17; if a value is converted to string with 17 digits of precision and then converted back to binary for some later calculation, the resulting binary value is guaranteed to be identical to the original one.

String Operations

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C *sprintf* format specifier **%d** for integers and **%g** for floating-point values. For example, the commands

```
expr {"0x03" > "2"} expr {"0y" < "0x12"}
```

both return 1. The first comparison is done using integer comparison, and the second is done using string comparison after the second operand is converted to the string ‘‘18’’.

A.16 file

Syntax

```
file option name ?arg arg ...?
```

Description

The **file** command provides several operations on a file's name or attributes. *name* is the name of a file; if it starts with a tilde, then tilde substitution is done before executing the command (see the manual entry for **Tcl_TildeSubst** for details). *Option* indicates what to do with the file name. Any unique abbreviation for *option* is acceptable. The valid options are:

file atime *name*

Returns a decimal string giving the time at which file *name* was last accessed. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its access time cannot be queried then an error is generated.

file dirname *name*

Returns all of the characters in *name* up to but not including the last slash character. If there are no slashes in *name* then returns ''.''. If the last slash in *name* is its first character, then return ''/''.

file executable *name*

Returns **1** if file *name* is executable by the current user, **0** otherwise.

file exists *name*

Returns **1** if file *name* exists and the current user has search privileges for the directories leading to it, **0** otherwise.

file extension *name*

Returns all of the characters in *name* after and including the last dot in *name*. If there is no dot in *name* then returns the empty string.

file isdirectory *name*

Returns **1** if file *name* is a directory, **0** otherwise.

file isfile *name*

Returns **1** if file *name* is a regular file, **0** otherwise.

file lstat *name varName*

Same as **stat** option (see below) except uses the *lstat* kernel call instead of *stat*. This means that if *name* refers to a symbolic link the information returned in *varName* is for the link rather than the

file it refers to. On systems that don't support symbolic links this option behaves exactly the same as the **stat** option.

file mtime *name*

Returns a decimal string giving the time at which file *name* was last modified. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its modified time cannot be queried then an error is generated.

file owned *name*

Returns **1** if file *name* is owned by the current user, **0** otherwise.

file readable *name*

Returns **1** if file *name* is readable by the current user, **0** otherwise.

file readlink *name*

Returns the value of the symbolic link given by *name* (i.e. the name of the file it points to). If *name* isn't a symbolic link or its value cannot be read, then an error is returned. On systems that don't support symbolic links this option is undefined.

file rootname *name*

Returns all of the characters in *name* up to but not including the last ''.'' character in the name. If *name* doesn't contain a dot, then returns *name*.

file size *name*

Returns a decimal string giving the size of file *name* in bytes. If the file doesn't exist or its size cannot be queried then an error is generated.

file stat *name varName*

Invokes the **stat** kernel call on *name*, and uses the variable given by *varName* to hold information returned from the kernel call. *VarName* is treated as an array variable, and the following elements of that variable are set: **atime**, **ctime**, **dev**, **gid**, **ino**, **mode**, **mtime**, **nlink**, **size**, **type**, **uid**. Each element except **type** is a decimal string with the value of the corresponding field from the **stat** return structure; see the manual entry for **stat** for details on the meanings of the values. The **type** element gives the type of the file in the same form returned by the command **file type**. This command returns an empty string.

file tail *name*

Returns all of the characters in *name* after the last slash. If *name* contains no slashes then returns *name*.

file type *name*

Returns a string giving the type of file *name*, which will be one of **file**, **directory**, **characterSpecial**, **blockSpecial**, **fifo**, **link**, or **socket**.

file writable *name*

Returns **1** if file *name* is writable by the current user, **0** otherwise.

A.17 flush

Syntax

```
flush fileId
```

Description

Flushes any output that has been buffered for *fileId. fileid* must have been the return value from a previous call to **open**, or it may be **stdout** or **stderr** to access one of the standard I/O streams; it must refer to a file that was opened for writing. The command returns an empty string.

A.18 for

Syntax

```
for start test next body
```

Description

for is a looping command, similar in structure to the C **for** statement. The *start*, *next*, and *body* arguments must be Tcl command strings, and *test* is an expression string. The **for** command first invokes the Tcl interpreter to execute *start*. Then it repeatedly evaluates *test* as an expression; if the result is non-zero it invokes the Tcl interpreter on *body*, then invokes the Tcl interpreter on *next*, then repeats the loop. The command terminates when *test* evaluates to 0. If a **continue** command is invoked within *body* then any remaining commands in the current execution of *body* are skipped; processing continues by invoking the Tcl interpreter on *next*, then evaluating *test*, and so on. If a **break** command is invoked within *body* or *next*, then the **for** command will return immediately. The operation of **break** and **continue** are similar to the corresponding statements in C. **For** returns an empty string.

A.19 foreach

Syntax

```
foreach varname list body
```

Description

In this command *varname* is the name of a variable, *list* is a list of values to assign to *varname*, and *body* is a Tcl script. For each element of *list* (in order from left to right), **foreach** assigns the contents of the field to *varname* as if the **lindex** command had been used to extract the field, then calls the Tcl interpreter to execute *body*. The **break** and **continue** statements may be invoked inside *body*, with the same effect as in the **for** command. **Foreach** returns an empty string.

A.20 format

Syntax

```
format formatString ?arg arg ...?
```

Description

This command generates a formatted string in the same way as the ANSI C **sprintf** procedure (it uses **sprintf** in its implementation). *FormatString* indicates how to format the result, using **%** conversion specifiers as in **sprintf**, and the additional arguments, if any, provide values to be substituted into the result. The return value from **format** is the formatted string.

Details On Formatting

The command operates by scanning *formatString* from left to right. Each character from the format string is appended to the result string unless it is a percent sign. If the character is a **%** then it is not copied to the result string. Instead, the characters following the **%** character are treated as a conversion specifier. The conversion specifier controls the conversion of the next successive *arg* to a particular format and the result is appended to the result string in place of the conversion specifier. If there are multiple conversion specifiers in the format string, then each one controls the conversion of one additional *arg*. The **format** command must be given enough *arg*s to meet the needs of all of the conversion specifiers in *formatString*.

Each conversion specifier may contain up to six different parts: an XPG3 position specifier, a set of flags, a minimum field width, a precision, a length modifier, and a conversion character. Any of these fields may be omitted except for the conversion character. The fields that are present must appear in the order given above. The paragraphs below discuss each of these fields in turn.

If the **%** is followed by a decimal number and a **$**, as in ‘‘**%2$d**’’, then the value to convert is not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where 1 corresponds to the first *arg*. If the conversion specifier requires multiple arguments because of ***** characters in the specifier then successive arguments are used, starting with the argument given by the number. This follows the XPG3 conventions for positional specifiers. If there are any positional specifiers in *formatString* then all of the specifiers must be positional.

The second portion of a conversion specifier may contain any of the following flag characters, in any order:

- **–** Specifies that the converted argument should be left-justified in its field (numbers are normally right-justified with leading spaces if needed).
- **+** Specifies that a number should always be printed with a sign, even if positive.

space Specifies that a space should be added to the beginning of the number if the first character isn't a sign.

0 Specifies that the number should be padded on the left with zeroes instead of spaces.

Requests an alternate output form. For **o** and **O** conversions it guarantees that the first digit is always **0**. For **x** or **X** conversions, **0x** or **0X** (respectively) will be added to the beginning of the result unless it is zero. For all floating-point conversions (**e**, **E**, **f**, **g**, and **G**) it guarantees that the result always has a decimal point. For **g** and **G** conversions it specifies that trailing zeroes should not be removed.

The third portion of a conversion specifier is a number giving a minimum field width for this conversion. It is typically used to make columns line up in tabular printouts. If the converted argument contains fewer characters than the minimum field width then it will be padded so that it is as wide as the minimum field width. Padding normally occurs by adding extra spaces on the left of the converted argument, but the **0** and **–** flags may be used to specify padding with zeroes on the left or with spaces on the right, respectively. If the minimum field width is specified as ***** rather than a number, then the next argument to the **format** command determines the minimum field width; it must be a numeric string.

The fourth portion of a conversion specifier is a precision, which consists of a period followed by a number. The number is used in different ways for different conversions. For **e**, **E**, and **f** conversions it specifies the number of digits to appear to the right of the decimal point. For **g** and **G** conversions it specifies the total number of digits to appear, including those on both sides of the decimal point (however, trailing zeroes after the decimal point will still be omitted unless the **#** flag has been specified). For integer conversions, it specifies a mimimum number of digits to print (leading zeroes will be added if necessary). For **s** conversions it specifies the maximum number of characters to be printed; if the string is longer than this then the trailing characters will be dropped. If the precision is specified with ***** rather than a number then the next argument to the **format** command determines the precision; it must be a numeric string.

The fourth part of a conversion specifier is a length modifier, which must be **h** or **l**. If it is **h** it specifies that the numeric value should be truncated to a 16-bit value before converting. This option is rarely useful. The **l** modifier is ignored.

The last thing in a conversion specifier is an alphabetic character that determines what kind of conversion to perform. The following conversion characters are currently supported:

d Convert integer to signed decimal string.

u Convert integer to unsigned decimal string.

i	Convert integer to signed decimal string; the integer may either be in decimal, in octal (with a leading **0**) or in hexadecimal (with a leading **0x**).
o	Convert integer to unsigned octal string.
x or **X**	Convert integer to unsigned hexadecimal string, using digits ''0123456789abcdef'' for **x** and ''0123456789ABCDEF'' for **X**).
c	Convert integer to the 8-bit character it represents.
s	No conversion; just insert string.
f	Convert floating-point number to signed decimal string of the form *xx.yyy*, where the number of *y*'s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output.
e or **e**	Convert floating-point number to scientific notation in the form *x.yyy***e**±*zz*, where the number of *y*'s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output. If the **E** form is used then **E** is printed instead of **e**.
g or **G**	If the exponent is less than –4 or greater than or equal to the precision, then convert floating-point number as for **%e** or **%E**. Otherwise convert as for **%f**. Trailing zeroes and a trailing decimal point are omitted.
%	No conversion: just insert **%**.

For the numerical conversions the argument being converted must be an integer or floating-point string; format converts the argument to binary and then converts it back to a string according to the conversion specifier.

Differences From Ansi Sprintf

The behavior of the format command is the same as the ANSI C **sprintf** procedure except for the following differences:

[1] **%p** and **%n** specifiers are not currently supported.

[2] For **%c** conversions the argument must be a decimal string, which will then be converted to the corresponding character value.

[3] The **l** modifier is ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the **l** modifier were present (i.e. type **double** is used for the internal representation). If the **h** modifier is specified then integer values are truncated to **short** before conversion.

A.21 gets

Syntax

```
gets fileId ?varName?
```

Description

The **gets** command reads the next line from the file given by *fileId* and discards the terminating newline character. If *varName* is specified then the line is placed in the variable by that name and the return value is a count of the number of characters read (not including the newline). If the end of the file is reached before reading any characters then –1 is returned and *varName* is set to an empty string. If *varName* is not specified then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so **eof** may have to be used to determine what really happened. If the last character in the file is not a newline character then **gets** behaves as if there were an additional newline character at the end of the file. *fileid* must be **stdin** or the return value from a previous call to **open**; it must refer to a file that was opened for reading.

Any existing end-of-file or error condition on the file is cleared at the beginning of the **gets** command.

A.22 glob

Syntax

```
glob ?switches? pattern ?pattern ...?
```

Description

The command performs file name ‘‘globbing’’ in a fashion similar to the csh shell. It returns a list of the files whose names match any of the *pattern* arguments.

If the initial arguments to **glob** start with **–** then they are treated as switches. The following switches are currently supported:

–nocomplain	Allows an empty list to be returned without error; without this switch an error is returned if the result list would be empty.
– –	Marks the end of switches. The argument following this one will be treated as a *pattern* even if it starts with a **–.**

The *pattern* arguments may contain any of the following special characters:

?	Matches any single character.
*****	Matches any sequence of zero or more characters.
[*chars***]**	Matches any single character in *chars*. If *chars* contains a sequence of the form *a–b* then any character between *a* and *b* (inclusive) will match.
\x	Matches the character *x*.
{*a***,***b***,...}**	Matches any of the strings *a*, *b*, etc.

As with csh, a ‘‘.’’ at the beginning of a file’s name or just after a ‘‘/’’ must be matched explicitly or with a {} construct. In addition, all ‘‘/’’ characters must be matched explicitly.

If the first character in a *pattern* is ‘‘~’’ then it refers to the home directory for the user whose name follows the ‘‘~’’. If the ‘‘~’’ is followed immediately by ‘‘/’’ then the value of the HOME environment variable is used.

The **glob** command differs from csh globbing in two ways. First, it does not sort its result list (use the **lsort** command if you want the list sorted).

Second, **glob** only returns the names of files that actually exist; in csh no check for existence is made unless a pattern contains a ?, *, or [] construct.

A.23 global

Syntax

```
global varname ?varname ...?
```

Description

The command is ignored unless a Tcl procedure is being interpreted. If so then it declares the given *varname*'s to be global variables rather than local ones. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varname*s will refer to the global variable by the same name.

A.24 history

Syntax

```
history ?option? ?arg arg ...?
```

Description

The **history** command performs one of several operations related to recently-executed commands recorded in a history list. Each of these recorded commands is referred to as an ‘‘event’’. When specifying an event to the **history** command, the following forms may be used:

[1] A number: if positive, it refers to the event with that number (all events are numbered starting at 1). If the number is negative, it selects an event relative to the current event (**–1** refers to the previous event, **–2** to the one before that, and so on).

[2] A string: selects the most recent event that matches the string. An event is considered to match the string either if the string is the same as the first characters of the event, or if the string matches the event in the sense of the **string match** command.

The **history** command can take any of the following forms:

history
Same as **history info**, described below.

history add *command* ?**exec**?
Adds the *command* argument to the history list as a new event. If **exec** is specified (or abbreviated) then the command is also executed and its result is returned. If **exec** isn’t specified then an empty string is returned as result.

history change *newValue* ?*event*?
Replaces the value recorded for an event with *newValue*. *Event* specifies the event to replace, and defaults to the *current* event (not event **–1**). This command is intended for use in commands that implement new forms of history substitution and wish to replace the current event (which invokes the substitution) with the command created through substitution. The return value is an empty string.

history event ?*event*?
Returns the value of the event given by *event*. *Event* defaults to **–1**. This command causes history revision to occur: see below for details.

history info ?*count*?

Returns a formatted string (intended for humans to read) giving the event number and contents for each of the events in the history list except the current event. If *count* is specified then only the most recent *count* events are returned.

history keep *count*

This command may be used to change the size of the history list to *count* events. Initially, 20 events are retained in the history list. This command returns an empty string.

history nextid

Returns the number of the next event to be recorded in the history list. It is useful for things like printing the event number in command-line prompts.

history redo ?*event*?

Re-executes the command indicated by *event* and return its result. *Event* defaults to **–1**. This command results in history revision: see below for details.

history substitute *old new* ?*event*?

Retrieves the command given by *event* (**–1** by default), replace any occurrences of *old* by *new* in the command (only simple character equality is supported; no wild cards), execute the resulting command, and return the result of that execution. This command results in history revision: see below for details.

history words *selector* ?*event*?

Retrieves from the command given by *event* (**–1** by default) the words given by *selector*, and return those words in a string separated by spaces. The **selector** argument has three forms. If it is a single number then it selects the word given by that number (**0** for the command name, **1** for its first argument, and so on). If it consists of two numbers separated by a dash, then it selects all the arguments between those two. Otherwise **selector** is treated as a pattern; all words matching that pattern (in the sense of **string match**) are returned. In the numeric forms **$** may be used to select the last word of a command. For example, suppose the most recent command in the history list is

```
format  {%s is %d years old} Alice
      [expr $ageInMonths/12]
```

Below are some history commands and the results they would produce:

```
history words $ [expr $ageInMonths/12]
history words 1-2 {%s is %d years  old} Alice
history words *a*o* {%s is %d years old} \
      [expr $ageInMonths/12]
```

History words results in history revision: see below for details.

History Revision

The history options **event**, **redo**, **substitute**, and **words** result in ‘‘history revision’’. When one of these options is invoked then the current event is modified to eliminate the history command and replace it with the result of the history command. For example, suppose that the most recent command in the history list is

```
set a [expr $b+2]
```

and suppose that the next command invoked is one of the ones on the left side of the table below. The command actually recorded in the history event will be the corresponding one on the right side of the table.

```
history redo set a [expr $b+2]
history s a b set b [expr $b+2]
set c [history w 2] set c [expr $b+2]
```

History revision is needed because event specifiers like **–1** are only valid at a particular time: once more events have been added to the history list a different event specifier would be needed.

History revision occurs even when **history** is invoked indirectly from the current event (e.g. a user types a command that invokes a Tcl procedure that invokes **history**): the top-level command whose execution eventually resulted in a **history** command is replaced.

If you wish to invoke commands like **history words** without history revision, you can use **history event** to save the current history event and then use **history change** to restore it later.

A.25 if

Syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 \
     elseif ... ?else? ?bodyN?
```

Description

The **if** command evaluates *expr1* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then **body2** is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional ''noise words'' to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

A.26 incr

Syntax

```
incr varName ?increment?
```

Description

Increments the value stored in the variable whose name is *varName*. The value of the variable must be an integer. If *increment* is supplied then its value (which must be an integer) is added to the value of variable *varName*; otherwise 1 is added to *varName*. The new value is stored as a decimal string in variable *varName* and also returned as result.

A.27 info

Syntax

```
info option ?arg arg ...?
```

Description

The command provides information about various internals of the Tcl interpreter. The legal *option*'s (which may be abbreviated) are:

info args *procname*
: Returns a list containing the names of the arguments to procedure *procname*, in order. *Procname* must be the name of a Tcl command procedure.

info body *procname*
: Returns the body of procedure *procname*. *Procname* must be the name of a Tcl command procedure.

info cmdcount
: Returns a count of the total number of commands that have been invoked in this interpreter.

info commands ?*pattern*?
: If *pattern* isn't specified, returns a list of names of all the Tcl commands, including both the built-in commands written in C and the command procedures defined using the **proc** command. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info complete *command*
: Returns 1 if *command* is a complete Tcl command in the sense of having no unclosed quotes, braces, brackets or array element names, If the command doesn't appear to be complete then 0 is returned. This command is typically used in line-oriented input environments to allow users to type in commands that span multiple lines; if the command isn't complete, the script can delay evaluating it until additional lines have been typed to complete the command.

info default *procname arg varname*
: *Procname* must be the name of a Tcl command procedure and *arg* must be the name of an argument to that procedure. If *arg* doesn't have a default value then the command returns **0**. Otherwise it returns **1** and places the default value of *arg* into variable *varname*.

info exists *varName*

Returns **1** if the variable named *varName* exists in the current context (either as a global or local variable), returns **0** otherwise.

info globals ?*pattern*?

If *pattern* isn't specified, returns a list of all the names of currently-defined global variables. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info level ?*number*?

If *number* is not specified, this command returns a number giving the stack level of the invoking procedure, or 0 if the command is invoked at top-level. If *number* is specified, then the result is a list consisting of the name and arguments for the procedure call at level *number* on the stack. If *number* is positive then it selects a particular stack level (1 refers to the top-most active procedure, 2 to the procedure it called, and so on); otherwise it gives a level relative to the current level (0 refers to the current procedure, -1 to its caller, and so on). See the **uplevel** command for more information on what stack levels mean.

info library

Returns the name of the library directory in which standard Tcl scripts are stored. The default value for the library is compiled into Tcl, but it may be overridden by setting the TCL_LIBRARY environment variable. If there is no TCL_LIBRARY variable and no compiled-in value then and error is generated. See the **library** manual entry for details of the facilities provided by the Tcl script library. Normally each application will have its own application-specific script library in addition to the Tcl script library; I suggest that each application set a global variable with a name like **$***app***_library** (where *app* is the application's name) to hold the location of that application's library directory.

info locals ?*pattern*?

If *pattern* isn't specified, returns a list of all the names of currently-defined local variables, including arguments to the current procedure, if any. Variables defined with the **global** and **upvar** commands will not be returned. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info patchlevel

Returns a decimal integer giving the current patch level for Tcl.

The patch level is incremented for each new release or patch, and it uniquely identifies an official version of Tcl.

info procs ?*pattern*?

If *pattern* isn't specified, returns a list of all the names of Tcl command procedures. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info script

If a Tcl script file is currently being evaluated (i.e. there is a call to **Tcl_EvalFile** active or there is an active invocation of the **source** command), then this command returns the name of the innermost file being processed. Otherwise the command returns an empty string.

info tclversion

Returns the version number for this version of Tcl in the form *x.y*, where changes to *x* represent major changes with probable incompatibilities and changes to *y* represent small enhancements and bug fixes that retain backward compatibility.

info vars ?*pattern*?

If *pattern* isn't specified, returns a list of all the names of currently-visible variables, including both locals and currently-visible globals. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

A.28 join

Syntax

```
join list ?joinString?
```

Description

The *list* argument must be a valid Tcl list. This command returns the string formed by joining all of the elements of *list* together with *joinString* separating each adjacent pair of elements. The *joinString* argument defaults to a space character.

A.29 lappend

Syntax

lappend *varName value ?value value ...?*

Description

The command treats the variable given by *varName* as a list and appends each of the *value* arguments to that list as a separate element, with spaces between elements. If *varName* doesn't exist, it is created as a list with elements given by the *value* arguments. **Lappend** is similar to **append** except that the *value*s are appended as list elements rather than raw text. This command provides a relatively efficient way to build up large lists. For example, ''**lappend a $b**'' is much more efficient than ''**set a [concat $a [list $b]]**'' when **$a** is long.

A.30 library

Syntax

```
auto_execok cmd
auto_load cmd
auto_mkindex dir pattern pattern ...
auto_reset
parray arrayName
unknown cmd ?arg arg ...?
```

Description

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

To access the procedures in the Tcl library, an application should source the file **init.tcl** in the library, for example with the Tcl command

```
source [info library]/init.tcl
```

This will define the unknown procedure and arrange for the other procedures to be loaded on-demand using the auto-load mechanism defined below.

Command Procedures

The following procedures are provided in the Tcl library:

auto_execok *cmd*

Determines whether there is an executable file by the name *cmd*. This command examines the directories in the current search path (given by the PATH enviornment variable) to see if there is an executable file named *cmd* in any of those directories. If so, it returns 1; if not it returns 0. **Auto_exec** remembers information about previous searches in an array named **auto_execs**; this avoids the path search in future calls for the same *cmd*. The command **auto_reset** may be used to force **auto_execok** to forget its cached information.

auto_load *cmd*

This command attempts to load the definition for a Tcl command named *cmd*. To do this, it searches an *auto-load path*, which is a list of one or more directories. The auto-load path is given by the

global variable **$auto_path** if it exists. If there is no **$auto_path** variable, then the TCLLIBPATH environment variable is used, if it exists. Otherwise the auto-load path consists of just the Tcl library directory. Within each directory in the auto-load path there must be a file **tclIndex** that describes one or more commands defined in that directory and a script to evaluate to load each of the commands. The **tclIndex** file should be generated with the **auto_mkindex** command. If *cmd* is found in an index file, then the appropriate script is evaluated to create the command.

The **auto_load** command returns 1 if *cmd* was successfully created. The command returns 0 if there was no index entry for *cmd* or if the script didn't actually define *cmd* (e.g. because index information is out of date). If an error occurs while processing the script, then that error is returned. **Auto_load** only reads the index information once and saves it in the array **auto_index**; future calls to **auto_load** check for *cmd* in the array rather than re-reading the index files. The cached index information may be deleted with the command **auto_reset**. This will force the next **auto_load** command to reload the index database from disk.

auto_mkindex *dir pattern pattern ...*

Generates an index suitable for use by **auto_load**. The command searches *dir* for all files whose names match any of the *pattern* arguments (matching is done with the **glob** command), generates an index of all the Tcl command procedures defined in all the matching files, and stores the index information in a file named **tclIndex** in *dir*. For example, the command

```
auto_mkindex foo *.tcl
```

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

Auto_mkindex parses the Tcl scripts in a relatively unsophisticated way: if any line contains the word **proc** as its first characters then it is assumed to be a procedure definition and the next word of the line is taken as the procedure's name. Procedure definitions that don't appear in this way (e.g. they have spaces before the **proc**) will not be indexed.

auto_reset

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

parray *arrayName*

Prints on standard output the names and values of all the elements in the array *arrayName*. **ArrayName** must be an array accessible to the caller of **parray**. It may be either local or global.

unknown *cmd* ?*arg arg ...*?

This procedure is invoked automatically by the Tcl interpreter whenever the name of a command doesn't exist. The **unknown** procedure receives as its arguments the name and arguments of the missing command.

Unknown first calls **auto_load** to load the command. If this succeeds, then it executes the original command with its original arguments. If the auto-load fails then **unknown** calls **auto_execok** to see if there is an executable file by the name *cmd*. If so, it invokes the Tcl **exec** command with *cmd* and all the *args* as arguments. If *cmd* can't be auto-executed, **unknown** checks to see if the command was invoked at top-level and outside of any script. If so, then **unknown** takes takes two additional steps. First, it sees if *cmd* has one of the following three forms: **!!**, **!***event*, or ^*old*^*new*?^?. If so, then **unknown** carries out history substitution in the same way that **csh** would for these constructs. Second, and last, **unknown** checks to see if *cmd* is a unique abbreviation for an existing Tcl command. If so, it expands the command name and executes the command with the original arguments. If none of the above efforts has been able to execute the command, **unknown** generates an error return. If the global variable **auto_noload** is defined, then the auto-load step is skipped. If the global variable **auto_noexec** is defined then the auto-exec step is skipped. Under normal circumstances the return value from **unknown** is the return value from the command that was eventually executed.

Variables

The following global variables are defined or used by the procedures in the Tcl library:

auto_execs

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

auto_index

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

auto_noexec

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

auto_noload

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

auto_path

If set, then it must contain a valid Tcl list giving directories to search during auto-load operations.

env(TCL_LIBRARY)

If set, then it specifies the location of the directory containing library scripts (the value of this variable will be returned by the command **info library**). If this variable isn't set then a default value is used.

env(TCLLIBPATH)

If set, then it must contain a valid Tcl list giving directories to search during auto-load operations. This variable is only used if **auto_path** is not defined.

unknown_active

This variable is set by **unknown** to indicate that it is active. It is used to detect errors where **unknown** recurses on itself infinitely. The variable is unset before **unknown** returns.

A.31 lindex

Syntax

```
lindex list index
```

Description

This command treats *list* as a Tcl list and returns the *index*'th element from it (0 refers to the first element of the list). In extracting the element, *lindex* observes the same rules concerning braces and quotes and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If *index* is negative or greater than or equal to the number of elements in *value*, then an empty string is returned.

A.32 linsert

Syntax

```
linsert list index element ?element element ...?
```

Description

The **linsert** command produces a new list from *list* by inserting all of the *element* arguments just before the *index*th element of *list*. Each *element* argument will become a separate element of the new list. If *index* is less than or equal to zero, then the new elements are inserted at the beginning of the list. If *index* is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

A.33 list

Syntax

```
list ?arg arg ...?
```

Description

The command returns a list comprised of all the *arg*s, or an empty string if no *arg*s are specified.

Braces and backslashes get added as necessary, so that the **index** command may be used on the result to re-extract the original arguments, and also so that **eval** may be used to execute the resulting list, with *arg1* comprising the command's name and the other *arg*s comprising its arguments. **List** produces slightly different results than **concat**: **concat** removes one level of grouping before forming the list, while **list** works directly from the original arguments. For example, the command

```
list a b {c d e} {f {g h}}
```

will return

```
a b {c d e} {f {g h}}
```

while **concat** with the same arguments will return

```
a b c d e f {g h}
```

A.34 llength

Syntax

```
llength list
```

Description

The **llength** command treats ***list*** as a list and returns a decimal string giving the number of elements in it.

A.35 lrange

Syntax

```
lrange list first last
```

Description

List must be a valid Tcl list. This command will return a new list consisting of elements *first* through *last*, inclusive. *Last* may be **end** (or any abbreviation of it) to refer to the last element of the list. If *first* is less than zero, it is treated as if it were zero. If *last* is greater than or equal to the number of elements in the list, then it is treated as if it were **end**. If *first* is greater than *last* then an empty string is returned. Note: ‘‘**lrange** *list first first*’’ does not always produce the same result as ‘‘**lindex** *list first*’’ (although it often does for simple fields that aren’t enclosed in braces); it does, however, produce exactly the same results as ‘‘**list [lindex** *list first***]**’’

A.36 lreplace

Syntax

```
lreplace list first last ?element element ...?
```

Description

Lreplace returns a new list formed by replacing one or more elements of *list* with the *element* arguments. *First* gives the index in *list* of the first element to be replaced. If *first* is less than zero then it refers to the first element of *list*; the element indicated by *first* must exist in the list. *Last* gives the index in *list* of the last element to be replaced; it must be greater than or equal to *first*. *Last* may be **end** (or any abbreviation of it) to indicate that all elements between *first* and the end of the list should be replaced. The *element* arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each *element* argument will become a separate element of the list. If no *element* arguments are specified, then the elements between *first* and *last* are simply deleted.

A.37 lsearch

Syntax

```
lsearch ?mode? list pattern
```

Description

The command searches the elements of *list* to see if one of them matches *pattern*. If so, the command returns the index of the first matching element. If not, the command returns **–1**.

The *mode* argument indicates how the elements of the list are to be matched against *pattern* and it must have one of the following values:

–exact	The list element must contain exactly the same string as *pattern*.
–glob	*Pattern* is a glob-style pattern which is matched against each list element using the same rules as the **string match** command.
–regexp	*Pattern* is treated as a regular expression and matched against each list element using the same rules as the **regexp** command.

If *mode* is omitted then it defaults to **–glob**.

A.38 lsort

Syntax

```
lsort ?switches? list
```

Description

The command sorts the elements of *list*, returning a new list in sorted order. By default ASCII sorting is used with the result returned in increasing order. However, any of the following switches may be specified before *list* to control the sorting process (unique abbreviations are accepted):

–ascii	Use string comparison with ASCII collation order. This is the default.
–integer	Convert list elements to integers and use integer comparison.
–real	Convert list elements to floating-point values and use floating comparison.
–command *command*	Use *command* as a comparison command. To compare two elements, evaluate a Tcl script consisting of *command* with the two elements appended as additional arguments. The script should return an integer less than, equal to, or greater than zero if the first element is to be considered less than, equal to, or greater than the second, respectively.
–increasing	Sort the list in increasing order (‘‘smallest’’ items first). This is the default.
–decreasing	Sort the list in decreasing order (‘‘largest’’ items first).

A.39 open

Syntax

```
open fileName ?access? ?permissions?
```

Description

The command opens a file and returns an identifier, ***fileId***, that may be used in future invocations of commands like **read**, **puts**, and **close**. *filename* gives the name of the file to open; if it starts with a tilde then tilde substitution is performed as described for **Tcl_TildeSubst**. If the first character of *fileName* is ''|'' then the remaining characters of *fileName* are treated as a command pipeline to invoke, in the same style as for **exec**. In this case, the identifier returned by **open** may be used to write to the command's input pipe or read from its output pipe.

The *access* argument indicates the way in which the file (or command pipeline) is to be accessed.

It may take two forms, either a string in the form that would be passed to the **fopen** library procedure or a list of POSIX access flags. It defaults to ''**r**''. In the first form *access* may have any of the following values:

r	Open the file for reading only; the file must already exist.
r+	Open the file for both reading and writing; the file must already exist.
w	Open the file for writing only. Truncate it if it exists. If it doesn't exist, create a new file.
w+	Open the file for reading and writing. Truncate it if it exists. If it doesn't exist, create a new file.
a	Open the file for writing only. The file must already exist, and the file is positioned so that new data is appended to the file.
a+	Open the file for reading and writing. If the file doesn't exist, create a new empty file. Set the initial access position to the end of the file.

In the second form, *access* consists of a list of any of the following flags, all of which have the standard POSIX meanings. One of the flags must be either **RDONLY**, **WRONLY** or **RDWR**.

RDONLY	Open the file for reading only.
WRONLY	Open the file for writing only.
RDWR	Open the file for both reading and writing.
APPEND	Set the file pointer to the end of the file prior to each write.
CREAT	Create the file if it doesn't already exist (without this flag it is an error for the file not to exist).

EXCL	If **CREAT** is specified also, an error is returned if the file already exists.
NOCTTY	If the file is a terminal device, this flag prevents the file from becoming the controlling terminal of the process.
NONBLOCK	Prevents the process from blocking while opening the file. For details refer to your system documentation on the **open** system call's **O_NONBLOCK** flag.
TRUNC	If the file exists it is truncated to zero length.

If a new file is created as part of opening it, *permissions* (an integer) is used to set the permissions for the new file in conjunction with the process's file mode creation mask. *Permissions* defaults to 0666.

If a file is opened for both reading and writing then **seek** must be invoked between a read and a write, or vice versa (this restriction does not apply to command pipelines opened with **open**). When *fileName* specifies a command pipeline and a write-only access is used, then standard output from the pipeline is directed to the current standard output unless overridden by the command. When *fileName* specifies a command pipeline and a read-only access is used, then standard input from the pipeline is taken from the current standard input unless overridden by the command.

A.40 pid

Syntax

```
pid ?fileId?
```

Description

If the ***fileId*** argument is given then it should normally refer to a process pipeline created with the **open** command. In this case the **pid** command will return a list whose elements are the process identifiers of all the processes in the pipeline, in order. The list will be empty if *fileId* refers to an open file that isn't a process pipeline. If no *fileId* argument is given then **pid** returns the process identifier of the current process. All process identifiers are returned as decimal strings.

A.41 proc

Syntax

```
proc name args body
```

Description

The **proc** command creates a new Tcl procedure named *name*, replacing any existing command or procedure there may have been by that name. Whenever the new command is invoked, the contents of *body* will be executed by the Tcl interpreter. *Args* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value.

When *name* is invoked a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name **args**, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to **args** are combined into a list (as if the **list** command had been used); this combined value is assigned to the local variable **args**.

When *body* is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the **global** command or the **upvar** command.

The **proc** command returns an empty string. When a procedure is invoked, the procedure's return value is the value specified in a **return** command. If the procedure doesn't execute an explicit **return**, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure-as-a-whole will return that same error.

A.42 puts

Syntax

```
puts ?–nonewline? ?fileId? string
```

Description

Writes the characters given by *string* to the file given by *fileId*. *FileId* must have been the return value from a previous call to **open**, or it may be **stdout** or **stderr** to refer to one of the standard I/O channels; it must refer to a file that was opened for writing. If no *fileId* is specified then it defaults to **stdout**. **Puts** normally outputs a newline character after *string*, but this feature may be suppressed by specifying the **–nonewline** switch. Output to files is buffered internally by Tcl; the **flush** command may be used to force buffered characters to be output.

A.43 pwd

Syntax

```
pwd
```

Description

Returns the path name of the current working directory.

A.44 read

Syntax

```
read ?–nonewline? fileId
read fileId numBytes
```

Description

In the first form, all of the remaining bytes are read from the file given by *fileId*; they are returned as the result of the command. If the **–nonewline** switch is specified then the last character of the file is discarded if it is a newline. In the second form, the extra argument specifies how many bytes to read; exactly this many bytes will be read and returned, unless there are fewer than *numBytes* bytes left in the file; in this case, all the remaining bytes are returned. ***fileid*** must be **stdin** or the return value from a previous call to **open**; it must refer to a file that was opened for reading. Any existing end-of-file or error condition on the file is cleared at the beginning of the **read** command.

A.45 regexp

Syntax

```
regexp ?switches? exp string ?matchVar? \
     ?subMatchVar subMatchVar ...?
```

Description

regexp determines whether the regular expression ***exp*** matches part or all of ***string*** and returns 1 if it does, 0 if it doesn't.

If additional arguments are specified after *string* then they are treated as the names of variables in which to return information about which part(s) of *string* matched *exp*. *MatchVar* will be set to the range of *string* that matched all of *exp*. The first *subMatchVar* will contain the characters in *string* that matched the leftmost parenthesized subexpression within *exp*, the next *subMatchVar* will contain the characters that matched the next parenthesized subexpression to the right in *exp*, and so on.

If the initial arguments to **regexp** start with **–** then they are treated as switches. The following switches are currently supported:

–nocase	Causes upper-case characters in *string* to be treated as lower case during the matching process.
–indices	Changes what is stored in the *subMatchVar*s. Instead of storing the matching characters from **string**, each variable will contain a list of two decimal strings giving the indices in *string* of the first and last characters in the matching range of characters.
– –	Marks the end of switches. The argument following this one will be treated as *exp* even if it starts with a **–.**

If there are more *subMatchVar*'s than parenthesized subexpressions within *exp*, or if a particular subexpression in *exp* doesn't match the string (e.g. because it was in a portion of the expression that wasn't matched), then the corresponding *subMatchVar* will be set to ''**–1 –1**'' if **–indices** has been specified or to an empty string otherwise.

Regular Expressions

Regular expressions are implemented using Henry Spencer's package, and much of the description of regular expressions below is copied verbatim from his manual entry.

A regular expression is zero or more *branches*, separated by ''|''. It matches anything that matches one of the branches.

A branch is zero or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by ‘‘*’’, ‘‘+’’, or ‘‘?’’. An atom followed by ‘‘*’’ matches a sequence of 0 or more matches of the atom. An atom followed by ‘‘+’’ matches a sequence of 1 or more matches of the atom. An atom followed by ‘‘?’’ matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a *range* (see below), ‘‘.’’ (matching any single character), ‘‘^’’ (matching the null string at the beginning of the input string), ‘‘$’’ (matching the null string at the end of the input string), a ‘‘\’’ followed by a single character (matching that character), or a single character with no other significance (matching that character).

A *range* is a sequence of characters enclosed in ‘‘[]’’. It normally matches any single character from the sequence. If the sequence begins with ‘‘^’’, it matches any single character *not* from the rest of the sequence. If two characters in the sequence are separated by ‘‘–’’, this is shorthand for the full list of ASCII characters between them (e.g. ‘‘[0-9]’’ matches any decimal digit). To include a literal ‘‘]’’ in the sequence, make it the first character (following a possible ‘‘^’’). To include a literal ‘‘–’’, make it the first or last character.

Choosing Among Alternative Matches

In general there may be more than one way to match a regular expression to an input string. For example, consider the command

```
regexp  (a*)b*  aabaaabb  x  y
```

Considering only the rules given so far**, x** and **y** could end up with the values **aabb** and **aa**, **aaab** and **aaa**, **ab** and **a**, or any of several other combinations. To resolve this potential ambiguity **regexp** chooses among alternatives using the rule ‘‘first then longest’’. In other words, it consders the possible matches in order working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

[1] If a regular expression could match two different parts of an input string then it will match the one that begins earliest.

[2] If a regular expression contains | operators then the leftmost matching sub-expression is chosen.

[3] In *****, **+**, and **?** constructs, longer matches are chosen in preference to shorter ones.

[4] In sequences of expression components the components are considered from left to right.

In the example from above, **(a*)b*** matches **aab**: the **(a*)** portion of the pattern is matched first and it consumes the leading **aa**; then the **b*** portion of the pattern consumes the next **b**. Or, consider the following example:

```
regexp  (ab|a)(b*)c  abc  x  y  z
```

After this command **x** will be **abc**, **y** will be **ab**, and **z** will be an empty string. Rule 4 specifies that **(ab|a)** gets first shot at the input string and Rule 2 specifies that the **ab** sub-expression is checked before the **a** sub-expression. Thus the **b** has already been claimed before the **(b*)** component is checked and **(b*)** must match an empty string.

A.46 regsub

Syntax

```
regsub ?switches? exp string subSpec varName
```

Description

The **regsub** command perform substitutions based on regular expression pattern matching. **regsub** matches the regular expression *exp* against ***string***, and it copies ***string*** to the variable whose name is given by ***varName***. The command returns 1 if there is a match and 0 if there isn't. If there is a match, then while copying ***string*** to ***varName*** the portion of *string* that matched *exp* is replaced with ***subSpec***. If ***subSpec*** contains a ''&'' or ''\0'', then it is replaced in the substitution with the portion of *string* that matched *exp*. If *subSpec* contains a ''*n*'', where *n* is a digit between 1 and 9, then it is replaced in the substitution with the portion of ***string*** that matched the *n*-th parenthesized subexpression of ***exp***. Additional backslashes may be used in ***subSpec*** to prevent special interpretation of ''&'' or ''\0'' or ''*n*'' or backslash. The use of backslashes in ***subSpec*** tends to interact badly with the Tcl parser's use of backslashes, so it's generally safest to enclose ***subSpec*** in braces if it includes backslashes.

If the initial arguments to **regexp** start with **–** then they are treated as switches. The following switches are currently supported:

–all — All ranges in ***string*** that match ***exp*** are found and substitution is performed for each of these ranges. Without this switch only the first matching range is found and substituted. If **–all** is specified, then ''&'' and ''*n*'' sequences are handled for each substitution using the information from the corresponding match.

–nocase — Upper-case characters in ***string*** will be converted to lower-case before matching against ***exp***; however, substitutions specified by ***subSpec*** use the original unconverted form of ***string***.

– – — Marks the end of switches. The argument following this one will be treated as ***exp*** even if it starts with a **–.**

See the manual entry for **regexp** for details on the interpretation of regular expressions.

A.47 rename

Syntax

rename *oldName newName*

Description

The **rename** command enames or deletse a command. **rename** renames the command that used to be called ***oldName*** so that it is now called ***newName***. If ***newName*** is an empty string then ***oldName*** is deleted. **rename** returns an empty string as result.

A.48 return

Syntax

```
return ?-code code? ?-errorinfo info? \
     ?-errorcode code? ?string?
```

Description

The **return** command returns from a procedure. **return** returns immediately from the current procedure (or top-level command or **source** command), with ***string*** as the return value. If ***string*** is not specified then an empty string will be returned as result.

Exceptional Returns

In the usual case where the **–code** option isn't specified the procedure will return normally (its completion code will be TCL_OK). However, the **–code** option may be used to generate an exceptional return from the procedure. ***code*** may have any of the following values:

ok	Normal return: same as if the option is omitted.
error	Error return: same as if the **error** command were used to terminate the procedure, except for handling of **errorInfo** and **errorCode** variables (see below).
return	The current procedure will return with a completion code of TCL_RETURN, so that the procedure that invoked it will return also.
break	The current procedure will return with a completion code of TCL_BREAK, which will terminate the innermost nested loop in the code that invoked the current procedure.
continue	The current procedure will return with a completion code of TCL_CONTINUE, which will terminate the current iteration of the innermost nested loop in the code that invoked the current procedure.
value	***Value*** must be an integer; it will be returned as the completion code for the current procedure.

The **–code** option is rarely used. It is provided so that procedures that implement new control structures can reflect exceptional conditions back to their callers.

Two additional options, **–errorinfo** and **–errorcode**, may be used to provide additional information during error returns. These options are ignored unless ***code*** is **error**.

The **–errorinfo** option specifies an initial stack trace for the **errorInfo** variable; if it is not specified then the stack trace left in **errorInfo** will include the call to the

procedure and higher levels on the stack but it will not include any information about the context of the error within the procedure. Typically the ***info*** value is supplied from the value left in **errorInfo** after a **catch** command trapped an error within the procedure.

If the **–errorcode** option is specified then ***code*** provides a value for the **errorCode** variable. If the option is not specified then **errorCode** will default to **NONE**.

A.49 scan

Syntax

```
scan string format varName ?varName ...?
```

Introduction

The **scan** command parses fields from an input string in the same fashion as the ANSI C **sscanf** procedure and returns a count of the number of fields sucessfully parsed. ***string*** gives the input to be parsed and ***format*** indicates how to parse it, using **%** conversion specifiers as in **sscanf**. Each ***varName*** gives the name of a variable; when a field is scanned from ***string*** the result is converted back into a string and assigned to the corresponding variable.

Details On Scanning

scan operates by scanning ***string*** and ***formatString*** together. If the next character in ***formatString*** is a blank or tab then it is ignored. Otherwise, if it isn't a **%** character then it must match the next non-white-space character of ***string***. When a **%** is encountered in ***formatString***, it indicates the start of a conversion specifier. A conversion specifier contains three fields after the **%**: a *****, which indicates that the converted value is to be discarded instead of assigned to a variable; a number indicating a maximum field width; and a conversion character. All of these fields are optional except for the conversion character.

When **scan** finds a conversion specifier in ***formatString***, it first skips any white-space characters in ***string***. Then it converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to **scan**. The following conversion characters are supported:

d — The input field must be a decimal integer. It is read in and the value is stored in the variable as a decimal string.

o — The input field must be an octal integer. It is read in and the value is stored in the variable as a decimal string.

x — The input field must be a hexadecimal integer. It is read in and the value is stored in the variable as a decimal string.

c — A single character is read in and its binary value is stored in the variable as a decimal string. Initial white space is not skipped in this case, so the input field may be a white-space character. This conversion is different from the ANSI standard in that the input field always consists of a single character and no field width may be specified.

s — The input field consists of all the characters up to the next white-space character; the characters are copied to the variable.

e or **f** or **g** The input field must be a floating-point number consisting of an optional sign, a string of decimal digits possibly con taining a decimal point, and an optional exponent consisting of an **e** or **E** followed by an optional sign and a string of decimal digits. It is read in and stored in the variable as a floating-point string.

[*chars*] The input field consists of any number of characters in ***chars***. The matching string is stored in the variable. If the first character between the brackets is a **]** then it is treated as part of ***chars*** rather than the closing bracket for the set.

[^*chars*] The input field consists of any number of characters not in ***chars***. The matching string is stored in the variable. If the character immediately following the ^ is a **]** then it is treated as part of the set rather than the closing bracket for the set.

The number of characters read from the input for a conversion is the largest number that makes sense for that particular conversion (e.g. as many decimal digits as possible for **%d**, as many octal digits as possible for **%o**, and so on). The input field for a given conversion terminates either when a white-space character is encountered or when the maximum field width has been reached, whichever comes first. If a ***** is present in the conversion specifier then no variable is assigned and the next scan argument is not consumed.

Differences From Ansi Sscanf

The behavior of the **scan** command is the same as the behavior of the ANSI C **sscanf** procedure except for the following differences:

[1] **%p** and **%n** conversion specifiers are not currently supported.

[2] For **%c** conversions a single character value is converted to a decimal string, which is then assigned to the corresponding ***varName***; no field width may be specified for this conversion.

[3] The **l**, **h**, and **L** modifiers are ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the **l** modifier were present (i.e. type **double** is used for the internal representation).

A.50 seek

Syntax

```
seek fileId offset ?origin?
```

Description

Change the current access position for ***fileId***. ***fileid*** must have been the return value from a previous call to **open**, or it may be **stdin**, **stdout**, or **stderr** to refer to one of the standard I/O channels. The ***offset*** and ***origin*** arguments specify the position at which the next read or write will occur for ***fileId***. ***offset*** must be an integer (which may be negative) and ***origin*** must be one of the following:

start The new access position will be ***offset*** bytes from the start of the file.

current The new access position will be ***offset*** bytes from the current access position; a negative ***offset*** moves the access position backwards in the file.

end The new access position will be ***offset*** bytes from the end of the file. A negative ***offset*** places the access position before the end-of-file, and a positive ***offset*** places the access position after the end-of-file.

The ***origin*** argument defaults to **start**. This command returns an empty string.

A.51 set

Syntax

```
set varName ?value?
```

Description

The **set** command reads and writes variables. **set** returns the value of variable ***varName***. If ***value*** is specified, then set the value of ***varName*** to ***value***, creating a new variable if one doesn't already exist, and return its value. If ***varName*** contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise ***varName*** refers to a scalar variable. If no procedure is active, then ***varName*** refers to a global variable. If a procedure is active, then ***varName*** refers to a parameter or local variable of the procedure unless the ***global*** command has been invoked to declare ***varName*** to be global.

A.52 source

Syntax

`source` *`fileName`*

Description

The **source** command evaluate a file as a Tcl script. source reads the file ***fileName*** and pass the contents to the Tcl interpreter as a script to evaluate in the normal fashion. The return value from **source** is the return value of the last command executed from the file. If an error occurs in evaluating the contents of the file then **source** will return that error. If a **return** command is invoked from within the file then the remainder of the file will be skipped and the **source** command will return normally with the result from the **return** command. If ***fileName*** starts with a tilde, then it is tilde-substituted as described in the **Tcl_TildeSubst** manual entry.

A.53 split

Syntax

```
split string ?splitChars?
```

Description

The **split** command splits a string into a proper Tcl list. **split** returns a list created by splitting ***string*** at each character that is in the ***splitChars*** argument. Each element of the result list will consist of the characters from ***string*** that lie between instances of the characters in ***splitChars***. Empty list elements will be generated if ***string*** contains adjacent characters in ***splitChars***, or if the first or last character of ***string*** is in ***splitChars***. If ***splitChars*** is an empty string then each character of ***string*** becomes a separate element of the result list. ***splitchars*** defaults to the standard white-space characters. For example,

```
split "comp.unix.misc" .
```

returns

```
"comp unix misc"
```

and

```
split "Hello world" {}
```

returns

```
H e l l o { } w o r l d
```

A.54 string

Syntax

```
string option arg ?arg ...?
```

Description

Performs one of several string operations, depending on ***option***. The legal ***option***s (which may be abbreviated) are:

string compare ***string1 string2***
: Perform a character-by-character comparison of strings ***string1*** and ***string2*** in the same way as the C **strcmp** procedure. Return –1, 0, or 1, depending on whether ***string1*** is lexicographically less than, equal to, or greater than ***string2***.

string first ***string1 string2***
: Search ***string2*** for a sequence of characters that exactly match the characters in ***string1***. If found, return the index of the first character in the first such match within ***string2***. If not found, return –1.

string index ***string charIndex***
: Returns the ***charIndex***'th character of the ***string*** argument. A ***charIndex*** of 0 corresponds to the first character of the string. If ***charIndex*** is less than 0 or greater than or equal to the length of the string then an empty string is returned.

string last ***string1 string2***
: Search ***string2*** for a sequence of characters that exactly match the characters in ***string1***. If found, return the index of the first character in the last such match within ***string2***. If there is no match, then return –1.

string length ***string***
: Returns a decimal string giving the number of characters in ***string***.

string match ***pattern string***
: See if ***pattern*** matches ***string***; return 1 if it does, 0 if it doesn't. Matching is done in a fashion similar to that used by the C-shell. For the two strings to match, their contents must be identical except that the following special sequences may appear in ***pattern***:

  ***** Matches any sequence of characters in ***string***, including a null string.

  ? Matches any single character in ***string***.

[*chars*] Matches any character in the set given by ***chars***. If a sequence of the form *x–y* appears in ***chars***, then any character between *x* and *y*, inclusive, will match.

x Matches the single character *x*. This provides a way of avoiding the special interpretation of the characters ***?[]**\ in ***pattern***.

string range *string first last*

Returns a range of consecutive characters from ***string***, starting with the character whose index is ***first*** and ending with the character whose index is ***last***. An index of 0 refers to the first character of the string. ***last*** may be **end** (or any abbreviation of it) to refer to the last character of the string. If ***first*** is less than zero then it is treated as if it were zero, and if ***last*** is greater than or equal to the length of the string then it is treated as if it were **end**. If ***first*** is greater than ***last*** then an empty string is returned.

string tolower *string*

Returns a value equal to ***string*** except that all upper case letters have been converted to lower case.

string toupper *string*

Returns a value equal to ***string*** except that all lower case letters have been converted to upper case.

string trim *string* ?***chars***?

Returns a value equal to ***string*** except that any leading or trailing characters from the set given by ***chars*** are removed. If ***chars*** is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimleft *string* ?***chars***?

Returns a value equal to ***string*** except that any leading characters from the set given by ***chars*** are removed. If ***chars*** is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimright *string* ?***chars***?

Returns a value equal to ***string*** except that any trailing characters from the set given by ***chars*** are removed. If ***chars*** is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

A.55 switch

Syntax

```
switch ?options? string pattern body ?pattern body ...?
switch ?options? string {pattern body ?pattern body ...?}
```

Description

The **switch** command matches its ***string*** argument against each of the ***pattern*** arguments in order. As soon as it finds a ***pattern*** that matches ***string*** it evaluates the following ***body*** argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last ***pattern*** argument is **default** then it matches anything. If no ***pattern*** argument matches ***string*** and no default is given, then **switch** returns an empty string.

If the initial arguments to **switch** start with **–** then they are treated as options. The following options are currently supported:

–exact	Use exact matching when comparing ***string*** to a pattern. This is the default.
–glob	When matching ***string*** to the patterns, use glob-style matching (i.e. the same as implemented by the **string match** command).
–regexp	When matching ***string*** to the patterns, use regular expression matching (i.e. the same as implemented by the **regexp** command).
– –	Marks the end of options. The argument following this one will be treated as ***string*** even if it starts with a **–.**

Two syntaxes are provided for the ***pattern*** and ***body*** arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line switch commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the ***pattern*** arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

If a ***body*** is specified as ‘‘**–**’’ it means that the ***body*** for the next pattern should also be used as the body for this pattern (if the next pattern also has a body of ‘‘**–**’’ then the body after that is used, and so on). This feature makes it possible to share a single ***body*** among several patterns.

Below are some examples of **switch** commands:

```
switch abc a – b {format 1} abc {format 2} default {format 3}
```

will return 2,

```
switch  –regexp aaab {
      ^a.*b$ –
      b  {format 1}
      a*  {format 2}
      default  {format 3}
```

will return 1, and

```
switch  xyz {
      a
      –
      b
            {format 1}
      a*
            {format 2}
      default
            {format 3}
}
```

will return 3.

A.56 tclvars

Description

`tclvars – Variables used by Tcl`

The following global variables are created and managed automatically by the Tcl library. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users.

env

This variable is maintained by Tcl as an array whose elements are the environment variables for the process. Reading an element will return the value of the corresponding environment variable. Setting an element of the array will modify the corresponding environment variable or create a new one if it doesn't already exist. Unsetting an element of **env** will remove the corresponding environment variable. Changes to the **env** array will affect the environment passed to children by commands like **exec**. If the entire **env** array is unset then Tcl will stop monitoring **env** accesses and will not update environment variables.

errorCode

After an error has occurred, this variable will be set to hold additional information about the error in a form that is easy to process with programs. **errorCode** consists of a Tcl list with one or more elements. The first element of the list identifies a general class of errors, and determines the format of the rest of the list. The following formats for **errorCode** are used by the Tcl core; individual applications may define additional formats.

ARITH ***code msg***

This format is used when an arithmetic error occurs (e.g. an attempt to divide by zero in the **expr** command). ***code*** identifies the precise error and ***msg*** provides a human-readable description of the error. ***code*** will be either DIVZERO (for an attempt to divide by zero), DOMAIN (if an argument is outside the domain of a function, such as acos(–3)), IOVERFLOW (for integer overflow), OVERLFLOW (for a floating-point overflow), or UNKNOWN (if the cause of the error cannot be determined).

CHILDKILLED ***pid sigName msg***

This format is used when a child process has been killed because of a signal. The second element of **errorCode** will be the process's identifier (in decimal). The third element will be the symbolic name of the signal that caused the process to terminate; it will be one of the names from the include file signal.h, such as

SIGPIPE. The fourth element will be a short human-readable message describing the signal, such as ‘‘write on pipe with no readers’’ for **SIGPIPE**.

CHILDSTATUS ***pid code***

This format is used when a child process has exited with a non-zero exit status. The second element of **errorCode** will be the process’s identifier (in decimal) and the third element will be the exit code returned by the process (also in decimal).

CHILDSUSP ***pid sigName msg***

This format is used when a child process has been suspended because of a signal. The second element of **errorCode** will be the process’s identifier, in decimal. The third element will be the symbolic name of the signal that caused the process to suspend; this will be one of the names from the include file signal.h, such as **SIGTTIN**. The fourth element will be a short human-readable message describing the signal, such as ‘‘background tty read’’ for **SIGTTIN**.

NONE

This format is used for errors where no additional information is available for an error besides the message returned with the error. In these cases **errorCode** will consist of a list containing a single element whose contents are **NONE**.

POSIX ***errName msg***

If the first element of **errorCode** is **POSIX**, then the error occurred during a POSIX kernel call. The second element of the list will contain the symbolic name of the error that occurred, such as **ENOENT**; this will be one of the values defined in the include file errno.h. The third element of the list will be a human-readable message corresponding to ***errName***, such as ‘‘no such file or directory’’ for the **ENOENT** case.

To set **errorCode**, applications should use library procedures such as **Tcl_SetErrorCode** and **Tcl_PosixError**, or they may invoke the **error** command. If one of these methods hasn’t been used, then the Tcl interpreter will reset the variable to **NONE** after the next error.

errorInfo

After an error has occurred, this string will contain one or more lines identifying the Tcl commands and procedures that were being executed when the most recent error occurred. Its contents take the form of a stack trace showing the various nested Tcl commands that had been invoked at the time of the error.

tcl_precision

If this variable is set, it must contain a decimal number giving the number of significant digits to include when converting floating-point values to strings. If this variable is not set then 6 digits are included. 17 digits is ‘‘perfect’’ for IEEE floating-point in that it allows double-precision values to be converted to strings and back to binary with no loss of precision.

A.57 tell

Syntax

```
tell fileId
```

Description

The **tell** command returns the current access position for an open file. Returns a decimal string giving the current access position in ***fileId***. ***fileid*** must have been the return value from a previous call to **open**, or it may be **stdin**, **stdout**, or **stderr** to refer to one of the standard I/O channels.

A.58 time

Syntax

```
time script ?count?
```

Description

The **time** command times the execution of a script. **time** will call the Tcl interpreter ***count*** times to evaluate ***script*** (or once if ***count*** isn't specified). **time** then returns a string of the form

```
503 microseconds per iteration
```

which indicates the average amount of time required per iteration, in microseconds. Time is measured in elapsed time, not CPU time.

A.59 trace

Syntax

```
trace option ?arg arg ...?
```

Description

The **trace** command monitors variable accesses. **trace** causes Tcl commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented. The legal option's (which may be abbreviated) are:

trace variable *name ops command*

Arrange for ***command*** to be executed whenever variable ***name*** is accessed in one of the ways given by ***ops***. ***name*** may refer to a normal variable, an element of an array, or to an array as a whole (i.e. ***name*** may be just the name of an array, with no parenthesized index). If ***name*** refers to a whole array, then ***command*** is invoked whenever any element of the array is manipulated.

Ops indicates which operations are of interest, and consists of one or more of the following letters:

r	Invoke ***command*** whenever the variable is read.
w	Invoke ***command*** whenever the variable is written.
u	Invoke ***command*** whenever the variable is unset. Variables can be unset explicitly with the **unset** command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to ***command*** so that the actual command is as follows:

```
command name1 name2 op
```

name1 and ***name2*** give the name(s) for the variable being accessed: if the variable is a scalar then ***name1*** gives the variable's name and ***name2*** is an empty string; if the variable is an array element then ***name1*** gives the name of the array and name2 gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then ***name1*** gives the array name and ***name2*** is an empty string. ***op*** indicates what operation is being performed on the variable, and is one of **r**, **w**, or **u** as defined above.

command executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a Tcl procedure, then ***command*** will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If ***command*** invokes a procedure (which it normally does) then the procedure will have to use **upvar** or **uplevel** if it wishes to access the traced variable. Note also that ***name1*** may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the **upvar** command.

For read and write traces, ***command*** can modify the variable to affect the result of the traced operation. If ***command*** modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from ***command*** is ignored except that if it returns an error of any sort then the traced operation also returns an error with the same error message returned by the trace command (this mechanism can be used to implement read-only variables, for example). For write traces, ***command*** is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, ***command*** will have to restore the old value of the variable.

While ***command*** is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by ***command*** will occur directly, without invoking ***command*** (or any other traces) again. However, if ***command*** unsets the variable then unset traces will be invoked.

When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked. Any errors in unset traces are ignored.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the **trace vdelete** command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

This command returns an empty string.

trace vdelete ***name ops command***

If there is a trace set on variable ***name*** with the operations and command given by ***ops*** and ***command***, then the trace is removed, so that ***command*** will never again be invoked. Returns an empty string.

trace vinfo ***name***

Returns a list containing one element for each trace currently set on variable ***name***. Each element of the list is itself a list containing two elements, which are the ***ops*** and ***command*** associated with the trace. If ***name*** doesn't exist or doesn't have any traces set, then the result of the command will be an empty string.

A.60 unknown

Syntax

unknown *cmdName* ?*arg arg ...*?

Description

The **unknown** command is used to handle attempts to use non-existent commands. **unknown** doesn't actually exist as part of Tcl, but Tcl will invoke it if it does exist. If the Tcl interpreter encounters a command name for which there is not a defined command, then Tcl checks for the existence of a command named **unknown**. If there is no such command, then the interpreter returns an error. If **unknown** exists, then it is invoked with arguments consisting of the fully-substituted name and arguments for the original non-existent command. **unknown** typically does things like searching through library directories for a command procedure with the name ***cmdName***, or expanding abbreviated command names to full-length, or automatically executing unknown commands as sub-processes. In some cases (such as expanding abbreviations) **unknown** will change the original command slightly and then (re-)execute it. The result of **unknown** is used as the result for the original non-existent command.

A.61 unset

Syntax

```
unset name ?name name ...?
```

Description

The **unset** command removes one or more variables. Each ***name*** is a variable name, specified in any of the ways acceptable to the **set** command. If a ***name*** refers to an element of an array then that element is removed without affecting the rest of the array. If a ***name*** consists of an array name with no parenthesized index, then the entire array is deleted. The **unset** command returns an empty string as result. An error occurs if any of the variables doesn't exist, and any variables after the non-existent one are not deleted.

A.62 uplevel

Syntax

```
uplevel ?level? arg ?arg ...?
```

Description

The **uplevel** command executes a script in a different stack frame. All of the ***arg*** arguments are concatenated as if they had been passed to **concat**; the result is then evaluated in the variable context indicated by ***level***. **uplevel** returns the result of that evaluation.

If ***level*** is an integer then it gives a distance (up the procedure calling stack) to move before executing the command. If ***level*** consists of **#** followed by a number then the number gives an absolute level number. If *level* is omitted then it defaults to **1**. *Level* cannot be defaulted if the first *command* argument starts with a digit or **#**.

For example, suppose that procedure **a** was invoked from top-level, and that it called **b**, and that **b** called **c**. Suppose that **c** invokes the **uplevel** command. If *level* is **1** or **#2** or omitted, then the command will be executed in the variable context of **b**. If *level* is **2** or **#1** then the command will be executed in the variable context of **a**. If *level* is **3** or **#0** then the command will be executed at top-level (only global variables will be visible).

uplevel causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose **c** invokes the command

```
uplevel 1 {set x 43; d}
```

where d is another Tcl procedure. The **set** command will modify the variable **x** in **b**'s context, and **d** will execute at level 3, as if called from **b**. If it in turn executes the command

```
uplevel {set x 42}
```

then the **set** command will modify the same variable x in b's context: the procedure c does not appear to be on the call stack when **d** is executing. The command ''**info level**'' may be used to obtain the level of the current procedure.

uplevel makes it possible to implement new control constructs as Tcl procedures (for example, **uplevel** could be used to implement the **while** construct as a Tcl procedure).

A.63 upvar

Syntax

```
upvar ?level? otherVar myVar ?otherVar myVar ...?
```

Description

The **upvar** command creates a link to a variable in a different stack frame. **upvar** arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. *Level* may have any of the forms permitted for the **uplevel** command, and may be omitted if the first letter of the first *otherVar* isn't **#** or a digit (it defaults to **1**). For each *otherVar* argument, **upvar** makes the variable by that name in the procedure frame given by *level* (or at global level, if *level* is **#0**) accessible in the current procedure by the name given in the corresponding *myVar* argument. The variable named by *otherVar* need not exist at the time of the call; it will be created the first time *myVar* is referenced, just like an ordinary variable. **upvar** may only be invoked from within procedures.*MyVar* may not refer to an element of an array, but *otherVar* may refer to an array element. **upvar** returns an empty string.

upvar simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as Tcl procedures. For example, consider the following procedure:

```
proc add2 name {     upvar $name x     set x [expr $x+2] }
```

add2 is invoked with an argument giving the name of a variable, and it adds two to the value of that variable. Although **add2** could have been implemented using **uplevel** instead of **upvar**, **upvar** makes it simpler for **add2** to access the variable in the caller's procedure frame.

If an upvar variable is unset (e.g. **x** in **add2** above), the **unset** operation affects the variable it is linked to, not the upvar variable. There is no way to unset an upvar variable except by exiting the procedure in which it is defined. However, it is possible to retarget an upvar variable by executing another **upvar** command.

A.64 while

Syntax

```
while test body
```

Description

The **while** command executes a script repeatedly as long as a condition is met. **while** evaluates ***test*** as an expression (in the same way that **expr** evaluates its argument). The value of the expression must a proper boolean value; if it is a true value then ***body*** is executed by passing it to the Tcl interpreter. Once ***body*** has been executed then ***test*** is evaluated again, and the process repeats until eventually ***test*** evaluates to a false boolean value. **continue** commands may be executed inside ***body*** to terminate the current iteration of the loop, and **break** commands may be executed inside ***body*** to cause immediate termination of the **while** command. **while** always returns an empty string.

Appendix B: Extended Tcl (TclX) Extensions

This Appendix contains hard copy versions of the manual pages for extensions that were added to Tcl 7.0 by Extended Tcl (TclX 7.0a). This Appendix is included solely as a convenience to the user.

> **NOTE —** Due to changes between TclX releases, some of the extensions in this Appendix may be outdated. To see a complete set of the current TclX extensions, use the **tclhelp** facility. (See Section 4.1.1 and Appendix B.14.)

B.1 Introduction

These extensions provide extend Tcl's capabilities by adding new commands to it, without changing the syntax of standard Tcl. Extended Tcl is a superset of standard Tcl and is built alongside the standard Tcl sources. Extended Tcl has three basic functional areas: A set of new commands, a Tcl shell (i.e., a UNIX shell-style command line and interactive environment), and a user-extensible library of useful Tcl procedures, any of which can be automatically loaded on the first attempt to execute it.

The command descriptions are separated by category, as shown in Table B-1.

Table B-1. TclX Categories

Category	Location
General Commands	Section B.3, Page B–3
Debugging and Development Commands	Section B.4, Page B–7
Unix Access Commands	Section B.5, Page B–10
File I/O Commands	Section B.6, Page B–20
TCP/IP Server Access	Section B.7, Page B–28
File Scanning Commands	Section B.8, Page B–31
Math Commands	Section B.9, Page B–34
List Manipulation Commands	Section B.10, Page B–36
Keyed Lists	Section B.11, Page B–40
String and Character Manipulation Commands	Section B.12, Page B–42
XPG/3 Message Catalog Commands	Section B.13, Page B–47
Help Facility	Section B.14, Page B–49
Tcl Loadable Libraries and Packages	Section B.15, Page B–50
Package Library Management Commands	Section B.16, Page B–52

B.2 fileName vs. fileId

In this section, the terms ***fileName*** and ***fileId*** are used. While they seem similar, possibly synonymous, they are very different.

fileName is the name of a file in the operating system, and ***fileId*** is an identifier Tcl creates when you open ***fileName*** using the **open** (Section A.39) command; ***fileId*** is a handle to the open ***fileName***. For example, assume you have a file called *Login_script1* and you want to open it. You could type the following:

```
open Login_script1
file3
```

Login_script1 is a ***fileName*** and **file3** is a ***fileId***. As with all handles, you can assign a ***fileId*** to a variable, such as in

```
set x [open Login_script1]
file3
```

B.3 General Commands

A set of general, useful Tcl commands, includes a command to begin an interactive session with Tcl, a facility for tracing execution, and a looping command.

B.3.1 dirs

Syntax

```
dirs
```

Description

This procedure lists the directories in the directory stack.

B.3.2 echo

Syntax

```
echo ?str ...?
```

Description

Writes zero or more strings to standard output, followed by a newline.

B.3.3 for_array_keys

Syntax

```
for_array_keys var array_name code
```

Description

This procedure performs a foreach-style loop for each key in the named array. The **break** and **continue** statements work as with **foreach**.

B.3.4 for_recursive_glob

Syntax

```
for_recursive_glob var dirlist globlist code
```

Description

This procedure performs a foreach-style loop over recursively matched files. All directories in *dirlist* are recursively searched (breadth-first), comparing each file found against the file glob patterns in **globlist**. For each matched file, the variable *var* is set to the file path and *code* is evaluated. Symbolic links are not followed.

B.3.5 infox

Syntax

```
infox option
```

Description

Return information about Extended Tcl, or the current application. The following **infox** command options are available:

version	Return the version number of Extended Tcl. The version number for Extended Tcl is generated by combining the base version of the standard Tcl code with a letter indicating the version of Extended Tcl being used. This is the documentation for version **7.0a**.
patchlevel	Return the patchlevel for Extended Tcl.
have_flock	Return **1** if the **flock** command is defined, **0** if it is not available.
have_fsync	Return **1** if the **fsync** system call is available and the **sync** command will sync individual files. **0** if it is not available and the **sync** command will always sync all file buffers.
have_msgcats	Return **1** if XPG message catalogs are available, **0** if they are not. The **catgets** is designed to continue to function without message catalogs, always returning the default string.
have_posix_signals	Return **1** if Posix signals are available (**block** and **unblock** options available for the signal command). **0** is returned if Posix signals are not available.
have_sockets	Return **1** if sockets are available (**server_*** suite and **fstat localhost** and **remotehost** options). **0** is returned if sockets are not available.

appname	Return the symbolic application name of the current application linked with the Extended Tcl library. The C variable **tclAppName** must be set by the application to return an application specific value for this variable.
applongname	Return a natural language name for the current application. The C variable **tclLongAppName** must be set by the application to return an application specific value for this variable.
appversion	Return the version number for the current application. The C variable **tclAppVersion** must be set by the application to return an application-specific value for this variable.
apppatchlevel	Return the patchlevel for the current application. The C variable **tclAppPatchlevel** must be set by the application to return an application-specific value for this variable.

B.3.6 loop

Syntax

```
loop var first limit ?increment? body
```

Description

loop is a looping command, similar in behavior to the Tcl **for** statement, except that the **loop** statement achieves substantially higher performance and is easier to code when the beginning and ending values of a loop are known, and the loop variable is to be incremented by a known, fixed amount every time through the loop.

The *var* argument is the name of a Tcl variable that will contain the loop index. The loop index is set to the value specified by *first*. The Tcl interpreter is invoked upon *body* zero or more times, where *var* is incremented by *increment* every time through the loop, or by one if *increment* is not specified. *Increment* can be negative in which case the loop will count downwards.

When *var* reaches *limit*, the loop terminates without a subsequent execution of *body*. For instance, if the original **loop** parameters would cause **loop** to terminate, say *first* was one, *limit* was zero and *increment* was not specified or was non-negative, *body* is not executed at all and **loop** returns.

The *first, limit* and *increment* are integer expressions. They are only evaluated once at the beginning of the loop.

If a **continue** command is invoked within *body* then any remaining commands in the current execution of *body* are skipped, as in the **for** command. If a **break** command is invoked within *body* then the **loop** command will return immediately. **Loop** returns an empty string.

B.3.7 popd

Syntax

popd

Description

This procedure pops the top directory entry from the directory stack and make it the current directory.

B.3.8 pushd

Syntax

`pushd ?dir?`

Description

This procedure pushs the current directory onto the directory stack and **cd** to the specified directory. If the directory is not specified, then the current directory is pushed, but remains unchanged.

B.3.9 recursive_glob

Syntax

`recursive_glob dirlist globlist`

Description

This procedure returns a list of recursively matches files. All directories in *dirlist* are recursively searched (breadth-first), comparing each file found against the file glob patterns in **globlist**. Symbolic links are not followed.

B.3.10 showproc

Syntax

`showproc ?procname ...?`

Description

This procedure lists the definition of the named procedures. Loading them if it is not already loaded. If no procedure names are supplied, the definitions of all currently loaded procedures are returned.

B.4 Debugging and Development Commands

This section contains information on commands and procedures that are useful for developing and debugging Tcl scripts.

B.4.1 cmdtrace

Syntax

```
cmdtrace level|on ?noeval? ?notruncate? ?procs? ?fileid?
cmdtrace off
cmdtrace depth
```

Description

Print a trace statement for all commands executed at depth of *level* or below (1 is the top level). If **on** is specified, all commands at any level are traced. The following options are available:

noeval Causes arguments to be printed unevaluated. If **noeval** is specified, the arguments are printed before evaluation. Otherwise, they are printed afterwards.

If the command line is longer than 60 characters, it is truncated to 60 and a "..." is postpended to indicate that there was more output than was displayed. If an evaluated argument contains a space, the entire argument will be enclosed inside of braces ('{}') to allow the reader to visually separate the arguments from each other.

notruncate Disables the truncation of commands and evaluated arguments.

procs Enables the tracing of procedure calls only. Commands that aren't procedure calls (i.e. calls to commands that are written in C, C++ or some object-compatible language) are not traced if the **procs** option is specified. This option is particularly useful for greatly reducing the output of **cmdtrace** while debugging.

fileid This is a file id as returned by the **open** command. If specified, then the trace output will be written to the file rather than stdout. A stdio buffer flush is done after every line is written so that the trace may be monitored externally or provide useful information for debugging problems that cause core dumps.

The most common use of this command is to enable tracing to a file during the development. If a failure occurs, a trace is then available when needed. Command tracing will slow down the execution of code, so it should be removed when code is

debugged. The following command will enable tracing to a file for the remainder of the program:

```
cmdtrace on [open cmd.log w]
```

When using the **off** argument, **cmdtrace** turns off all tracing.

When using the **depth** argument, **cmdtrace** returns the current maximum trace level, or zero if trace is disabled.

B.4.2 edprocs

Syntax

```
edprocs ?proc...?
```

Description

This procedure writes the named procedures, or all currently defined procedures, to a temporary file, then calls an editor on it (as specified by the **EDITOR** environment variable, or **vi** if none is specified), then sources the file back in if it was changed.

B.4.3 profile

Syntax

```
profile ?-commands? on
profile off arrayVar
```

Description

This command is used to collect a performance profile of a Tcl script. It collects data at the Tcl procedure level. The number of calls to a procedure, and the amount of real and CPU time is collected. Time is also collected for the global context. The procedure data is collected by bucketing it based on the procedure call stack, this allows determination of how much time is spent in a particular procedure in each of it's calling contexts.

The **on** option enables profile data collection. If the **-commands** option is specified, data on all commands within a procedure is collected as well a procedures. Multiple occurrences of a command within a procedure are not distinguished, but this data may still be useful for analysis.

The **off** option turns off profiling and moves the data collected to the array *arrayVar*. The array is address by a list containing the procedure call stack. Element zero is the top of the stack, the procedure that the data is for. The data in each entry is a list consisting of the procedure call count and the real time and CPU time in milliseconds

spent in the procedure (and all procedures it called). The list is in the form {*count real cpu*}. A Tcl procedure **profrep** is supplied for reducing the data and producing a report

B.4.4 profrep

Syntax

```
profrep profDataVar sortKey ?outFile? ?userTitle?
```

Description

This procedure generates a report from data collect from the profile command. **ProfDataVar** is the name of the array containing the data returned by the **profile** command. **SortKey** indicates which data value to sort by. It should be one of "**calls**", "**cpu**" or "**real**". **OutFile** is the name of file to write the report to. If omitted, stdout is assumed. **UserTitle** is an optional title line to add to output.

B.4.5 saveprocs

Syntax

```
saveprocs fileName ?proc...?
```

Description

This procedure saves the definition of the named procedure, or all currently defined procedures if none is specified, to the named file.

B.5 UNIX Access Commands

These commands provide access to many basic Unix facilities, including process handling, date and time processing, signal handling, linking and unlinking files, setting file, process, and user attributes, and the executing commands via the shell.

B.5.1 chgrp

Syntax

`chgrp` *`group filelist`*

Description

Set the group id of each file in the list *filelist* to *group*, which can be either a group name or a numeric group id.

B.5.2 chmod

Syntax

`chmod` *`mode filelist`*

Description

Set permissions of each of the files in the list *filelist* to *mode*, where *mode* is an absolute numeric mode or symbolic permissions as in the UNIX **chmod(1)** command. To specify a mode as octal, it should be prefixed with a "0" (e.g. 0622).

B.5.3 chroot

Syntax

`chroot` *`dirname`*

Description

Change root directory to *dirname*, by invoking the POSIX **chroot(2)** system call. This command only succeeds if running as `root`.

B.5.4 chown

Syntax

```
chown owner|{owner group} filelist
```

Description

Set owner of each file in the list *filelist* to *owner*, which can be a user name or numeric user id. If the first parameter is a list, then the owner is set to the first element of the list and the group is set to the second element. *Group* can be a group name or numeric group id. If *group* is {}, then the file group will be set to the login group of the specified user.

B.5.5 convertclock

Syntax

```
convertclock dateString ?GMT|{}? ?baseClock?
```

Description

Convert *dateString* to an integer clock value (see **getclock**). This command can parse and convert virtually any standard date and/or time string, which can include standard time zone mnemonics. If only a time is specified, the current date is assumed. If the string does not contain a time zone mnemonic, the local time zone is assumed, unless the **GMT** argument is specified, in which case the clock value is calculated assuming that the specified time is relative to Greenwich Mean Time. If *baseClock* is specified, it is taken as the current clock value. This is useful for determining the time on a specific day.

The character string consists of zero or more specifications of the following form: *time* - A time of day, which is of the form *hh*[*:mm*[*:ss*]] [*meridian*] [*zone*] or *hhmm* [*meridian*] [*zone*]. If no meridian is specified, *hh* is interpreted on a 24-hour clock. *date* - A specific month and day with optional year. The acceptable formats are *mm/dd*[*/yy*], *yyyy/mm/dd*, *monthnamedd*[,*yy*], *ddmonthname*[*yy*"], and *da y,ddmonthnameyy*.

The default year is the current year. If -year is provided, xmyDate returns the current year as a four character string.

> **NOTE —** If a timestamp contains a two character year, xmyDate determines the century based on wether the year is in the range 70-99 or the range 00-38. Any year in the range 70-99 is assumed to mean 1970-1999, and any year in the range of 00-38 is assumed to mean 2000-2038. Values in the range 39-69 are NOT supported.

.

relative time - A specification relative to the current time. The format is *numberunit*; acceptable units are *year*, *fortnight*, *month*, *week*, *day*, *hour*, *minute* (or *min*), and *second* (or *sec*). The unit can be specified as a singular or plural, as in *3weeks*. These modifiers may also be specified: *tomorrow*, *yesterday*, *today*, *now*, *last*, *this*, *next*, *ago*. The actual date is calculated according to the following steps. First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added. Next, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences.

convertclock ignores case when parsing all words. The names of the months and days of the week can be abbreviated to their first three letters, with optional trailing period. Periods are ignored in any timezone or meridian values.

Note that *convertclock* will convert symbolic time-zone names, but these are not standardized and there are conflicts with various parts of the world. Use GMT when trying to produce a portable time that can then be converted back to a numeric value. The only dates in the range 1902 and 2037 may be converted. Some examples are:

```
convertclock "14 Feb 92"
convertclock "Feb 14, 1992 12:20 PM PST"
convertclock "12:20 PM Feb 14, 1992"
```

B.5.6 fmtclock

Syntax

`fmtclock` *`clockval ?format? ?GMT|{}?`*

Description

Converts a Unix integer time value, typically returned by **getclock**, **convertclock**, or the **atime**, **mtime**, or **ctime** options of the **file** command, to human-readable form. The *format* argument is a string that describes how the date and time are to be formatted. Field descriptors consist of a "%'' followed by a field descriptor character. All other characters are copied into the result. Valid field descriptors are:

```
%% - Insert a %.
%a - Abbreviated weekday name.
%A - Full weekday name
%b - Abbreviated month name.
%B - Full month name.
%d - Day of month (01 - 31).
%D - Date as %m/%d/%y.
%e - Day of month (1-31), no leading zeros.
```

```
%h - Abbreviated month name.
%H - Hour (00 - 23).
%I - Hour (00 - 12).
%j - Day number of year (001 - 366).
%m - Month number (01 - 12).
%M - Minute (00 - 59).
%n - Insert a new line.
%p - AM or PM.
%r - Time as %I:%M:%S %p.
%R - Time as %H:%M.
%S - Seconds (00 - 59).
%t - Insert a tab.
%T - Time as %H:%M:%S.
%U - Week number of year (01 - 52), Sunday is the first
      day of the week.
%w - Weekday number (Sunday = 0).
%W - Week number of year (01 - 52), Monday is the first
      day of the week.
%x - Local specific date format.
%X - Local specific time format.
%y - Year within century (00 - 99).
%Y - Year as ccyy (e.g. 1990)
%Z - Time zone name.
```

If format is not specified, "%a %b %d %H:%M:%S %Z %Y" is used. If **GMT** is specified, the time will be formatted as Greenwich Mean Time. If the argument is not specified or is empty, then the local timezone will be used as defined by the TIMEZONE environment variable.

B.5.7 fork

Syntax

```
fork
```

Description

Fork the current Tcl process. Fork returns zero to the child process and the process number of the child to the parent process. If the fork fails, a Tcl error is generated.

If an **execl** is not going to be performed before the child process does output, or if a **close** and **dup** sequence is going to be performed on **stdout** or **stderr**, then a **flush** should be issued against **stdout**, **stderr** and any other open output file before doing the **fork**. Otherwise characters from the parent process pending in the buffers will be output by both the parent and child processes.

NOTE — If you are **forking** in a Tk based application you must **execl** before doing any window operations in the child or you will receive a **BadWindow** error from the X server.

B.5.8 getclock

Syntax

```
getclock
```

Description

Return the current date and time as a system-dependent integer value. The unit of the value is seconds, allowing it to be used for relative time calculations.

B.5.9 id

Syntax

```
id options
```

Description

This command provides a means of getting, setting and converting user, group and process ids. The **id** command has the following options:

id user ?*name*? **id userid** ?*uid*?	Set the real and effective user ID to *name* or *uid*, if the name (or uid) is valid and permissions allow it. If the name (or uid) is not specified, the current name (or uid) is returned.
id convert userid *uid* **id convert user** *name*	Convert a user ID number to a user name, or vice versa.
id group ?*name*? **id groupid** ?*gid*?	Set the real and effective group ID to *name* or *gid*, if the name (or gid) is valid and permissions allow it. If the group name (or gid) is not specified, the current group name (or gid) is returned.
id convert groupid *gid* **id convert group** *name*	Convert a group ID number to a group name, or vice versa.
id effective user **id effective userid**	Return the effective user name, or effective user ID number, respectively.
id effective group **id effective groupid**	Return the effective group name, or effective group ID number, respectively.
id host	Return the hostname of the system the program is running on.
id process	Return the process ID of the current process.
id process parent	Return the process ID of the parent of the current process.
id process group	Return the process group ID of the current process.
id process group set	Set the process group ID of the current process to its process ID.

B.5.10 kill

Syntax

```
kill ?-pgroup? ?signal? idlist
```

Description

Send a signal to the each process in the list ***idlist***, if permitted. ***signal***, if present, is the signal number or the symbolic name of the signal, see the **signal** system call manual page. The leading "SIG'' is optional when the signal is specified by its symbolic name. The default for ***signo*** is 15, SIGTERM.

If **-pgroup** is specified, the numbers in *idlist* are take as process group ids and the signal is sent to all of the process in that process group. A process group id of **0** specifies the current process group.

B.5.11 link

Syntax

```
link ?-sym? srcpath destpath
```

Description

Create a directory entry, *destpath*, linking it to the existing file, *srcpath*. If **-sym** is specified, a symbolic link, rather than a hard link, is created. (The **-sym** option is only available on systems that support symbolic links.)

B.5.12 mkdir

Syntax

```
mkdir ?-path? dirList
```

Description

Create each of the directories in the list *dirList*. The mode on the new directories is 777, modified by the umask. If **-path** is specified, then any non-existent parent directories in the specified path(s) are also created.

B.5.13 nice

Syntax

```
nice ?priorityincr?
```

Description

Change or return the process priority. If *priorityincr* is omitted, the current priority is returned. If *priorityincr* is positive, it is added to the current *priority* level, up to a system defined maximum (normally **19**).

Negative *priorityincr* values cumulatively increase the program's priority down to a system defined minimum (normally **-19**); increasing priority with negative niceness values will only work for the superuser.

The new priority is returned.

B.5.14 readdir

Syntax

```
readdir dirPath
```

Description

Returns a list containing the contents of the directory *dirPath*. The directory entries "." and ".." are not returned.

B.5.15 rmdir

Syntax

```
rmdir ?-nocomplain? dirList
```

Description

Remove each of the directories in the list *dirList*. If **-nocomplain** is specified, then errors will be ignored.

B.5.16 sleep

Syntax

```
sleep seconds
```

Description

The TclX command **sleep** has been redefined to be the same as the MYNAH **xmySleep** extension.

B.5.17 system

Syntax

```
system command
```

Description

Executes ***command*** via the *system*(3) call. Differs from **exec** in that **system** doesn't return the executed command's standard output as the result string, and **system** goes through the Unix shell to provide wildcard expansion, redirection, etc, as is normal from an **sh** command line. The exit code of the command is returned.

B.5.18 sync

Syntax

```
sync ?fileId?
```

Description

If *fileId* is not specified, or if it is and this system does not support the **fsync** system call, issues a **sync** system call to flush all pending disk output. If *fileId* is specified and the system does support the **fsync** system call, issues an *fsync* on the file corresponding to the specified Tcl *fileId* to force all pending output to that file out to the disk.

If *fileId* is specified, the file must be writable. A **flush** will be issued against the *fileId* before the sync.

The *infox have_fsync* command can be used to determine if "**sync** *fileId*" will do a *sync* or a *fsync*.

B.5.19 times

Syntax

```
times
```

Description

Return a list containing the process and child execution times in the form

```
utime stime cutime cstime
```

See also the **times(2)** system call manual page. The values are in milliseconds.

B.5.20 umask

Syntax

```
umask ?octalmask?
```

Description

Sets file-creation mode mask to the octal value of *octalmask*. If *octalmask* is omitted, the current mask is returned.

B.5.21 unlink

Syntax

```
unlink ?-nocomplain? filelist
```

Description

Delete (unlink) the files whose names are in the list *filelist*. If **-nocomplain** is specified, then errors will be ignored.

B.6 File I/O Commands

These commands extend the stdio-style file I/O capabilities present in Tcl and above. These extensions include searching ASCII-sorted data files, copying files, duplicating file descriptors, control of file access options, retrieving open file status, and creating pipes with the **pipe** system call. An interface to the **select** system call is available on Unix systems that support it.

It should be noted that Tcl file I/O is implemented on top of the stdio library. By default, the file is buffered. When communicating to a process through a pipe, a **flush** command should be issued to force the data out. Alternatively, the **fcntl** command may be used to set the buffering mode of a file to line-buffered or unbuffered.

> **NOTE —** Remember to use the basic Tcl **open** (Sections 4.9.1 and A.39) and **close** (Sections 4.9.2 and A.7) commands to perform the basic I/O functions.

B.6.1 bsearch

Syntax

```
bsearch fileId key ?retvar? ?compare_proc?
```

Description

Search an opened file *fileId* containing lines of text sorted into ascending order for a match. *Key* contains the string to match. If *retvar* is specified, then the line from the file is returned in *retvar*, and the command returns **1** if *key* was found, and **0** if it wasn't. If *retvar* is not specified or is a null name, then the command returns the line that was found, or an empty string if *key* wasn't found.

By default, the key is matched against the first white-space separated field in each line. The field is treated as an ASCII string. If *compare_proc* is specified, then it defines the name of a Tcl procedure to evaluate against each line read from the sorted file during the execution of the **bsearch** command. *Compare_proc* takes two arguments, the key and a line extracted from the file. The compare routine should return a number less than zero if the key is less than the line, zero if the key matches the line, or greater than zero if the key is greater than the line. The file must be sorted in ascending order according to the same criteria *compare_proc* uses to compare the key with the line, or erroneous results will occur.

B.6.2 copyfile

Syntax

```
copyfile ?-bytes num|-maxbytes num? fromFileId toFileId
```

Description

Copies the rest of the file specified by *fromFileId*, starting from its current position, to the file specified by *toFileId*, starting from its current position.

If **-bytes** is specified, then *num* bytes are copied. If less than *num* bytes are available, an error is returned. If **-maxbytes** is specified, then *num* bytes are copied but no error is returned if less are available.

The command returns the number of bytes that were copied.

The **-bytes** option is particularly useful for mixing binary data in with ASCII commands or data in a data stream.

B.6.3 dup

Syntax

```
dup fileId ?targetFileId?
```

Description

Duplicate an open file. A new file id is opened that addresses the same file as *fileId*. If *targetFileId* is specified, the file is dup to this specified file id. Normally this is **stdin**, **stdout**, or **stderr**. The dup command will handle flushing output and closing this file. The target file should be open if its one of **stdin**, **stdout**, or **stderr** and the process is not going to do an **execl**. Otherwise internal C code that uses one of these files via direct access to stdio FILE struct may behave strangely or fail.

B.6.4 fcntl

Syntax

`fcntl` *`fileId attribute ?value?`*

Description

This command either sets or clears a file option or returns its current value. If *value* is not specified, then the current value of *attribute* is returned. The following attributes may be specified:

RDONLY The file is opened for reading only. (Get only)

WRONLY The file is opened for writing only. (Get only)

RDWR The file is opened for reading and writing. (Get only)

READ If the file is readable. (Get only).

WRITE If the file is writable. (Get only).

APPEND The file is opened for append-only writes. All writes will be forced to the end of the file.

NONBLOCK The file is to be accessed with non-blocking I/O. See the **read** system call for a description of how it affects the behavior of file reads.

CLOEXEC Close the file on an process exec. If the **execl** command or some other mechanism causes the process to do an exec, the file will be closed if this option is set.

NOBUF The file is not buffered. If set, then there no stdio buffering for the file.

LINEBUF Output the file will be line buffered. The buffer will be flushed when a newline is written, when the buffer is full, or when input is requested.

The **APPEND**, **NONBLOCK**, and **CLOEXEC** attributes may be set or cleared by specifying the attribute name and a value **1** to set the attribute and **0** to clear it.

The **NOBUF** and **LINEBUF** attributes may only be set (a value of **1**) and only one of the options may be selected. Once set, it may not be changed. These options should be set before any I/O operations have been done on the file or data may be lost.

B.6.5 flock

Syntax

```
flock options fileId ?start? ?length? ?origin?
```

Description

This command places a lock on all or part of the file specified by ***fileId***. The lock is either advisory or mandatory, depending on the mode bits of the file. The lock is placed beginning at relative byte offset *start* for *length* bytes. If *start* or *length* is omitted or empty, zero is assumed. If *length* is zero, then the lock always extents to end of file, even if the file grows. If *origin* is "**start**", then the offset is relative to the beginning of the file. If it is "**current**", it is relative to the current access position in the file. If it is "**end**", then it is relative to the end-of-file (a negative is before the EOF, positive is after). If *origin* is omitted, **start** is assumed.

The following *options* are recognized:

-read Place a read lock on the file. Multiple processes may be accessing the file with read-locks.

-write Place a write lock on the file. Only one process may be accessing a file if there is a write lock.

-nowait If specified, then the process will not block if the lock can not be obtained. With this option, the command returns 1 if the lock is obtained and 0 if it is not.

See your system's **fcntl** system call documentation for full details of the behavior of file locking. If locking is being done on ranges of a file, it is best to use unbuffered file access (see the **fcntl** command).

B.6.6 for_file

Syntax

```
for_file var filename { code }
```

Description

This procedure implements a loop over the contents of a file. For each line in *filename*, it sets *var* to the line and executes *code*.

The **break** and **continue** commands work as with **foreach**.

For example, the command

```
for_file line /etc/passwd {echo $line}
```

would echo all the lines in the password file.

B.6.7 funlock

Syntax

`funlock` *`fileId ?start? ?length? ?origin?`*

Remove a locked from a file that was previously placed with the *flock* command. The arguments are the same as for the *flock* command, see that command for more details.

B.6.8 frename

Syntax

`frename` *`oldPath newPath`*

Description

Renames ***oldPath*** to ***newPath***. This command does not support renaming across file systems unless the **rename** system call supports them. If renaming across file systems is desired, **exec** the **mv** command.

B.6.9 fstat

Syntax

`fstat` *`fileId ?item?|?stat arrayvar?`*

Description

Obtain status information about an open file.

The following keys are used to identify data items:

atime	The time of last access.
ctime	The time of last file status change
dev	The device containing a directory for the file. This value uniquely identifies the file system that contains the file.
gid	The group ID of the file's group.
ino	The inode number. This field uniquely identifies the file in a given file system.
mode	The mode of the file (see the **mknod** system call).
mtime	Time when the data in the file was last modified.
nlink	The number of links to the file.

size The file size in bytes.

tty If the file is associated with a terminal, then 1 otherwise 0.

type The type of the file in symbolic form, which is one of the following values: **file**, **directory**, **characterSpecial**, **blockSpecial**, **fifo**, **link**, or **socket**.

uid The user ID of the file's owner.

If one of these keys is specified as *item*, then that data item is returned.

If **stat** *arrayvar* is specified, then the information is returned in the array *arrrayvar*. Each of the above keys indexes an element of the array containing the data.

If only *fileId* is specified, the command returns the data as a keyed list.

The following value may be returned only if explicitly asked for, it will not be returned with the array or keyed list forms:

localhost If *fileId* is a TCP/IP socket, then a list is returned with the first element being the local host IP address and the second element being the local host port number.

remotehost If *fileId* is a TCP/IP socket connection, then a list is returned with the first element being the remote host IP address and the second element being the remote host IP port number.

B.6.10 lgets

Syntax

```
lgets fileId ?varName?
```

Description

Reads the next Tcl list from the file given by *fileId* and discards the terminating newline character. This command differs from the **gets** command, in that it reads Tcl lists rather than lines. If the list contains a newline, then that newline will be returned as part of the result. Only a newline not quoted as part of the list indicates the end of the list. There is no corresponding command for outputting lists, as **puts** will do this correctly. If *varName* is specified, then the line is placed in the variable by that name and the return value is a count of the number of characters read (not including the newline). If the end of the file is reached before reading any characters then –1 is returned and *varName* is set to an empty string. If *varName* is not specified then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so **eof** may have to be used to determine what really happened.

B.6.11 pipe

Syntax

```
pipe ?fileId_var_r fileId_var_w?
```

Description

Create a pipe. If ***fileId_var_r*** and ***fileId_var_r*** are specified, then **pipe** will set the a variable named ***fileId_var_r*** to contain the fileId of the side of the pipe that was opened for reading, and ***fileId_var_w*** will contain the fileId of the side of the pipe that was opened for writing.

If the ***fileId*** variables are not specified, then a list containing the read and write fileIdw is returned as the result of the command.

B.6.12 read_file

Syntax

```
read_file ?-nonewline? fileName
read_file fileName numBytes
```

Description

This procedure reads the file *fileName* and returns the contents as a string. If **-nonewline** is specified, then the last character of the file is discarded if it is a newline. The second form specifies exactly how many bytes will be read and returned, unless there are fewer than *numBytes* bytes left in the file; in this case, all the remaining bytes are returned.

B.6.13 select

Syntax

```
select readfileIds ?writefileIds? ?exceptfileIds? ?timeout?
```

The **select** command allows an Extended Tcl program to wait on zero or more files being ready for reading, writing, have an exceptional condition pending, or for a timeout period to expire. ***readFileIds***, ***writeFileIds***, ***exceptFileIds*** are each lists of fileIds, as returned from open, to query. An empty list ({}) may be specified if a category is not used.

select takes the following options:

readFileIds Specifies a list of files that are checked to see if data is available for reading.

writeFileIds Specifies a list of files that are checked if the specified files are clear for writing.

exceptFileIds Specifies a list of files that are checked to see if an exceptional condition has occurred (typically, an error). The write and exception checking is most useful on devices, however, the read checking is very useful when communicating with multiple processes through pipes. **select** considers data pending in the stdio input buffer for read files as being ready for reading, the files do. not have to be unbuffered.

timeout Specifies (in seconds, a floating point timeout value. If an empty list is supplied (or the parameter is omitted), then no timeout is set. If the value is zero, then the select command functions as a poll of the files, returning immediately even if none are ready.

If the timeout period expires with none of the files becoming ready, then the command returns an empty list. Otherwise the command returns a list of three elements, each of those elements is a list of the fileIds that are ready in the read, write and exception classes. If none are ready in a class, then that element will be the null list.

Example

Entering

```
select {file3 file4 file5} {file6 file7} {} 10.5
```

could return

```
{file3 file4} {file6} {}
```

or perhaps

```
file3 {} {}
```

B.6.14 write_file

Syntax

`write_file` *`fileName string ?string...?`*

Description

This procedure writes the specified strings to the named file.

B.7 TCP/IP Server Access

Commands are provided to access TCP/IP-based servers, and to create them. It is easy to build servers using Extended Tcl that run under **inetd**, or even servers that run stand-alone and accept and manage multiple simultaneous connections. The **fstat remotehost** and **fstat localhost** requests are useful both for clients and servers.

B.7.1 server_accept

Syntax

```
server_accept ?options? fileid
```

Description

Accept a TCP/IP connection to the server socket associated with *fileid*. Option can be either **-buf** or **-nobuf**. Buffer options have the same effect as described in **server_open**.

B.7.2 server_create

Syntax

```
server_create ?options?
```

Description

Creates a TCP/IP server socket on the local machine. A file handle is returned upon successful creation. When a connection request is made to the server, the file handle becomes read-ready. Connections can be accepted using **server_accept**.
The file handle can be detected as read-ready using **select**, by using **fcntl** to make the handle nonblocking and then calling **server_accept**, or by using a file-status interface to *wish* such as Mark Diekhans' *addinput* package.

Options include **-myip** *ipNumber*, **-myport** *portNumber*, and **-backlog** *count*. If no IP number is specified, one is supplied by the system. If no port number is specified, a port number is assigned by the system.

Backlog defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, the client may receive an error with an indication of *ECONNREFUSED*, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed. Note that on at least some BSD-based systems the backlog is silently limited to 5, regardless of the value specified.

B.7.3 server_info

Syntax

```
server_info addresses hostname
server_info official_name hostname
server_info aliases_name hostname
```

Description

Optain information about a TCP/IP server.

If the **addresses** option is specified, the list of IP addresses for *hostname* is returned.

If the **official_name** option is specified, the official name for *hostname* is returned.

If the **aliases** option is specified, a list of aliases for *hostname* is returned. (Note that these are IP number aliases, not DNS *CNAME* aliases. See *ifconfig(2)*.)

B.7.4 server_open

Syntax

```
server_open ?option? host service
```

Description

Open a TCP/IP connection to a server of *host* on the port specified by *service*. The server is then accessed using the standard Tcl file I/O commands. *Host* may be a host name or an IP address. *Port* may be a port number of a service name. Options include **-buf**, **-nobuf**, **-myip** and **-myport**. If no buffer option is specified, **-buf** is the default. The **-buf** option specifies that the file is buffered. In this case, a pair of Tcl file ids are returned in a list. The first id is open for read access, the second for write. When writing to the file, the **flush** command must be used to force data in the buffer to be sent to the server. The **close** command must be called against both file ids when you are done using the socket. Buffered access will result in significantly better performance when reading data, and will also improve the performance of a series of writes done without intervening reads. The **fcntl** command may be used to make one of these files unbuffered, or line buffered.

If **-nobuf** is specified, then the file is unbuffered and a single file id, open for both reading and writing, is returned.

If **-myip** *ipNumber* is specified, it will define the IP number for your side of the connection. This is useful for multi-homed hosts (hosts with more than one IP address). Note that only IP addresses corresponding to network interfaces on your machine may be used. If **-myip** is not specified, the operating system will assign the IP number for you.

If **-myport** *portNumber* is specified, it will define the port number for your side of the connection. If the port number is already in use, an error will be returned. If the port number is in the privileged range, the Tcl program will have to be running as superuser, or an error will be returned.

If a destination host name is supplied and more that one address is valid for the host, the host's addresses will be tried in the order returned until one can be connected to, or the list is exhausted. You may also use the **server_info** command to obtain the list of valid address.

B.7.5 server_receive

Syntax

```
server_receive fileid length ?options?
```

Description

Receives from the TCP/IP socket associated with *fileid* a message of up to *length* bytes and returns it to the caller.

Options include *out_of_band* to receive out-of-band data, *peek* to peek at the incoming data (returning data from the receive queue without removing the data from the queue), *wait_all* to request that the operation block until the full request is satisfied, and *keepnewline* -- normally **server_receive** will discard the last character of the message if it is a newline.

B.7.6 server_send

Syntax

```
server_send fileid string ?options?
```

Description

Sends the specified string to the TCP/IP connection corresponding to *fileid.* **server_send** works like **puts**, including puts' *nonewline* option, but is better at detecting lost connections and other IP-related error conditions.

Other options include *dont_route*, which requests that routing be bypassed and the direct interface used (usually used only by diagnostic or routing programs), and *out_of_band*, which can be used to send out-of-band data on the socket.

B.8 File Scanning Commands

These commands provide a facility to scan files, matching lines of the file against regular expressions and executing Tcl code on a match. With this facility you can use Tcl to do the sort of file processing that is traditionally done with *awk*. And since Tcl's approach is more declarative, some of the scripts that can be rather difficult to write in awk are simple to code in Tcl.

File scanning in Tcl centers around the concept of a *scan context*. A scan context contains one or more match statements, which associate regular expressions to scan for with Tcl code to be executed when the expressions are matched.

B.8.1 scancontext

Syntax

```
scancontext ?option?
```

Description

This command manages file scan contexts. A scan context is a collection of regular expressions and commands to execute when that regular expression matches a line of the file. A context may also have a single default match, to be applied against lines that do not match any of the regular expressions. Multiple scan contexts may be defined and they may be reused on multiple files. A scan context is identified by a context handle. The **scancontext** command takes the following forms:

scancontext *create*

Create a new scan context. The **scanmatch** command is used to define patterns in the context. A contexthandle is returned, which the Tcl programmer uses to refer to the newly created scan context in calls to the Tcl file scanning commands.

scancontext *delete contexthandle*

Delete the scan context identified by *contexthandle*, and free all of the match statements and compiled regular expressions associated with the specified context.

B.8.2 scanfile

Syntax

```
scanfile ?-copyfile copyFileId? contexthandle fileId
```

Description

Scan the file specified by *fileId*, starting from the current file position. Check all patterns in the scan context specified by *contexthandle* against it, executing the match commands corresponding to patterns matched.

If the optional *-copyfile* argument is specified, the next argument is a file ID to which all lines not matched by any pattern (excluding the default pattern) are to be written.

B.8.3 scanmatch

Syntax

```
scanmatch ?-nocase? contexthandle ?regexp? commands
```

Description

Specify Tcl *commands*, to be evaluated when *regexp* is matched by a **scanfile** command. The match is added to the scan context specified by *contexthandle*. Any number of match statements may be specified for a give context. *Regexp* is a regular expression (see the **regexp** command). If **-nocase** is specified as the first argument, the pattern is matched regardless of alphabetic case.

If *regexp* is not specified, then a default match is specified for the scan context. The default match will be executed when a line of the file does not match any of the regular expressions in the current scancontext.

The array **matchInfo** is available to the Tcl code that is executed when an expression matches (or defaults). It contans information about the file being scanned and where within it the expression was matched.

matchInfo is local to the top level of the match command unless declared global at that level by the Tcl **global** command. If it is to be used as a global, it *must* be declared global before **scanfile** is called (since **scanfile** sets the **matchInfo** before the match code is executed, a subsequent **global** will override the local variable). The following array entries are available:

matchInfo(line)
: Contains the text of the line of the file that was matched.

matchInfo(offset)
: The byte offset into the file of the first character of the line that was matched.

matchInfo(linenum)
The line number of the line that was matched. This is relative to the first line scanned, which is usually, but not necessarily, the first line of the file. The first line is line number one.

matchInfo(handle)
he file id (handle) of the file currently being scanned.

matchInfo(copyHandle)
The file id (handle) of the file specified by the **-copyfile** option. The element does not exist if **-copyfile** was not specified.

matchInfo(submatch0)
Will contain the characters matching the first parenthesized subexpression. The second will be contained in **submatch1**, etc.

matchInfo(subindex0)
Will contain the a list of the starting and ending indices of the string matching the first parenthesized subexpression. The second will be contained in **subindex1**, etc.

All **scanmatch** patterns that match a line will be processed in the order in which their specifications were added to the scan context. The remainder of the **scanmatch** pattern-command pairs may be skipped for a file line if a **continue** is executed by the Tcl code of a preceding, matched pattern.

If a **return** is executed in the body of the match command, the **scanfile** command currently in progress returns, with the value passed to **return** as its return value.

B.9 Math Commands

Several extended math commands commands make many additional math functions available in TclX. In addition, a set of procedures provide command access to the math functions supported by the **expr** command.

The following procedures provide command interfaces to the expr math functions. They take the same arguments as the **expr** functions and may take expressions as arguments.

abs	**acos**	**asin**	**atan2**
atan	**ceil**	**cos**	**cosh**
double	**exp**	**floor**	**fmod**
hypot	**int**	**log10**	**log**
pow	**round**	**sin**	**sinh**
sqrt	**tan**	**tanh**	

B.9.1 max

Syntax

`max` *num1 num2 ?..numN?*

Description

Returns the argument that has the highest numeric value. The arguments, *numN* may be any interger or floating point values.

B.9.2 min

Syntax

`min` *num1 num2 ?..numN?*

Description

Returns the argument that has the lowest numeric value. The arguments, *numN* may be any interger or floating point values.

B.9.3 random

Syntax

```
random limit | seed ?seedval?
```

Description

Generate a pseudorandom integer number greater than or equal to zero and less than *limit*. If **seed** is specified, then the command resets the random number generator to a starting point derived from the **seedval**. This allows one to reproduce pseudorandom number sequences for testing purposes. If *seedval* is omitted, then the seed is set to a value based on current system state and the current time, providing a reasonably interesting and ever-changing seed.

B.10 List Maninipulation Commands

Extended Tcl provides additional list manipulation commands and procedures.

B.10.1 intersect

Syntax

```
intersect lista listb
```

Description

Procedure to return the logical intersection of two lists. The returned list will be sorted.

B.10.2 intersect3

Syntax

```
intersect3 lista listb
```

Description

Procedure to intersects two lists, returning a list containing three lists: The first list returned is everything in *lista* that wasn't in *listb*. The second list contains the intersection of the two lists, and the third list contains all the elements that were in *listb* but weren't in *lista*. The returned lists will be sorted.

B.10.3 lassign

Syntax

```
lassign list var ?var...?
```

Description

Assign successive elements of a list to specified variables. If there are more variable names than fields, the remaining variables are set to the empty string. If there are more elements than variables, a list of the unassigned elements is returned.

For example,

```
lassign {dave 100 200 {Dave Foo}} name uid gid longName
```

assigns *name* to ''dave'', *uid* to ''100'', *gid* to ''200'', and *longName* to ''Dave Foo''.

B.10.4 lempty

Syntax

```
lempty list
```

Description

Determine if the specified list is empty. If empty, 1 is returned, otherwise, 0 is returned. This command is an alternative to comparing a list to an empty string.

B.10.5 lmatch

Syntax

```
lmatch ?mode? list pattern
```

Description

Search the elements of *list*, returning a list of all elements matching *pattern*. If none match, an empty list is returned.

The *mode* argument indicates how the elements of the list are to be matched against *pattern* and it must have one of the following values:

–exact The list element must contain exactly the same string as *pattern*.

–glob *Pattern* is a glob-style pattern which is matched against each list element using the same rules as the **string match** command.

–regexp *Pattern* is treated as a regular expression and matched against each list element using the same rules as the **regexp** command.

If *mode* is omitted then it defaults to **–glob**.

B.10.6 lrmdups

Syntax

```
lrmdups list
```

Description

Procedure to remove duplicate elements from a list. The returned list will be sorted.

B.10.7 lvarcat

Syntax

```
lvarcat var string ?string...?
```

Description

This command treats each *string* argument as a list and concatenates them to the end of the contents of *var*, forming a a single list. The list is stored back into *var* and also returned as the result. if *var* does not exist, it is created.

B.10.8 lvarpop

Syntax

```
lvarpop var ?indexExpr? ?string?
```

Description

The **lvarpop** command pops (deletes) the element indexed by the expression *indexExpr* from the list contained in the variable *var*. If *index* is omitted, then 0 is assumed. If *string*, is specified, then the deleted element is replaced by *string*. The replaced or deleted element is returned. Thus ‘‘lvarpop argv 0’’ returns the first element of argv, setting argv to contain the remainder of the string.

If the expression *indexExpr* starts with the string **end**, then **end** is replaced with the index of the last element in the list. If the expression starts with **len**, then **len** is replaced with the length of the list.

B.10.9 lvarpush

Syntax

```
lvarpush var string ?indexExpr?
```

Description

The **lvarpush** command pushes (inserts) *string* as an element in the list contained in the variable *var*. The element is inserted before position *indexExpr* in the list. If *index* is omitted, then 0 is assumed. If *var* does not exists, it is created.

If the expression *indexExpr* starts with the string **end**, then **end** is replaced with the index of the last element in the list. If the expression starts with **len**, then **len** is replaced with the length of the list. Note the a value of **end** means insert the string before the last element.

B.10.10 union

Syntax

```
union lista listb
```

Description

Procedure to return the logical union of the two specified lists. Any duplicate elements are removed.

B.11 Keyed Lists

Extended Tcl defines a special type of list referred to as *keyed lists.* These lists provided a structured data type built upon standard Tcl lists. This provides a functionality similar to *struct*s in the C programming language.

A keyed list is a list in which each element contains a key and value pair. These element pairs are stored as lists themselves, where the key is the first element of the list, and the value is the second. The key-value pairs are refered to as *fields.* This is an example of a keyed list:

```
{{NAME {Frank Zappa}} {JOB {musician and composer}}}
```

If the variable **person** contained the above list, then **keylget person NAME** would return **{Frank Zappa}**. Executing the command:

```
keylset person ID 106
```

would make **person** contain

```
{{ID 106} {NAME {Frank Zappa}} {JOB {musician and composer}}
```

Fields may contain subfields; '.' is the seperator character. Subfields are actually fields where the value is another keyed list. Thus the following list has the top level fields *ID* and *NAME*, and subfields *NAME.FIRST* and *NAME.LAST*:

```
{ID 106} {NAME {{FIRST Frank} {LAST Zappa}}}
```

There is no limit to the recursive depth of subfields, allowing one to build complex data structures.

Keyed lists are constructed and accessed via a number of commands. All keyed list management commands take the name of the variable containing the keyed list as an argument (i.e. passed by reference), rather than passing the list directly.

B.11.1 keyldel

Syntax

```
keyldel listvar key
```

Description

Delete the field specified by ***key*** from the keyed list in the variable ***listvar***. This removes both the key and the value from the keyed list.

B.11.2 keylget

Syntax

```
keylget listvar ?key? ?retvar | {}?
```

Description

Return the value associated with ***key*** from the keyed list in the variable ***listvar***. If ***retvar*** is not specified, then the value will be returned as the result of the command. In this case, if ***key*** is not found in the list, an error will result.

If ***retvar*** is specified and *key* is in the list, then the value is returned in the variable ***retvar*** and the command returns **1** if the key was present within the list. If *key* isn't in the list, the command will return **0**, and ***retvar*** will be left unchanged. If **{}** is specified for ***retvar***, the value is not returned, allowing the Tcl programmer to determine if a key is present in a keyed list without setting a variable as a side-effect.

If ***key*** is omitted, then a list of all the keys in the keyed list is returned.

B.11.3 keylkeys

Syntax

```
keylkeys listvar ?key?
```

Description

Return the a list of the keyes in the keyed list in the variable ***listvar***. If ***keys*** is specified, then it is the name of a key field who's subfield keys are to be retrieve.

B.11.4 keylset

Syntax

```
keylset listvar key value ?key2 value2 ...?
```

Description

Set the value associated with ***key***, in the keyed list contained in the variable ***listvar***, to ***value***. If ***listvar*** does not exists, it is created. If ***key*** is not currently in the list, it will be added. If it already exists, ***value*** replaces the existing value. Multiple keywords and values may be specified, if desired.

B.12 String and Character Manipulation Commands

The commands provide additional functionality to classify characters, convert characters between character and numeric values, index into a string, determine the length of a string, extract a range of character from a string, replicate a string a number of times, and transliterate a string (similar to the Unix *tr* program).

B.12.1 cequal

Syntax

`cequal` *`string1 string2`*

Description

This command compares two strings for equality. Returns **1** if *string1* and *string2* are the identical and **0** if they are not. This command is a short-cut for **string compare** and avoids the problems with string expressions being treated unintentionally as numbers.

B.12.2 cexpand

Syntax

`cexpand` *`string`*

Description

Expand backslash sequences in *string* to their actual characters. No other substitution takes place.

B.12.3 cindex

Syntax

`cindex` *`string indexExpr`*

Description

Returns the character indexed by the expression *indexExpr* (zero based) from *string*. If the expression *indexExpr* starts with the string **end**, then **end** is replaced with the index of the last character in the string. If the expression starts with **len**, then **len** is replaced with the length of the string.

B.12.4 clength

Syntax

```
clength string
```

Description

Returns the length of *string* in characters. This command is a shortcut for:

```
string length string
```

B.12.5 crange

Syntax

```
crange string firstExpr lastExpr
```

Description

Returns a range of characters from *string* starting at the character indexed by the expression *firstExpr* (zero-based) until the character indexed by the expression *lastExpr*.

If the expression *firstExpr* or **lastExpr** starts with the string **end**, then **end** is replaced with the index of the last character in the string. If the expression starts with **len**, then **len** is replaced with the length of the string.

B.12.6 csubstr

Syntax

```
csubstr string firstExpr lengthExpr
```

Description

Returns a range of characters from *string* starting at the character indexed by the expression *firstExpr* (zero-based) for *lengthExpr* characters.

If the expression *firstExpr* or **lengthExpr** starts with the string **end**, then **end** is replaced with the index of the last character in the string. If the expression starts with **len**, then **len** is replaced with the length of the string.

B.12.7 ctoken

Syntax

```
ctoken strvar separators
```

Description

Parse a token out of a character string. The string to parse is contained in the variable named *strvar*. The string *separators* contains all of the valid separator characters for tokens in the string. All leading separators are skipped and the first token is returned. The variable *strvar* will be modified to contain the remainder of the string following the token.

B.12.8 ctype

Syntax

```
ctype ?-failindex var? class string
```

Description

ctype determines whether all characters in *string* are of the specified *class*. It returns **1** if they are all of *class*, and **0** if they are not, or if the string is empty. This command also provides another method (besides **format** and **scan**) of converting between an ASCII character and its numeric value. The following **ctype** commands are available:

ctype *?-failindex var? alnum string*
: Tests that all characters are alphabetic or numeric characters as defined by the character set.

ctype *?-failindex var? alpha string*
: Tests that all characters are alphabetic characters as defined by the character set.

ctype *?-failindex var? ascii string*
: Tests that all characters are an ASCII character (a non-negative number less than 0200).

ctype char *number*
: Converts the numeric value, *string*, to an ASCII character. Number must be in the range 0 through 255.

ctype *?-failindex var? cntrl string*
: Tests that all characters are ‘‘control characters’’ as defined by the character set.

ctype *?-failindex var? digit string*
: Tests that all characters are valid decimal digits, i.e. 0 through 9.

ctype *?-failindex var? graph string*
Tests that all characters within are any character for which *ctype print* is true, except for space characters.

ctype *?-failindex var? lower string*
Tests that all characters are lowercase letters as defined by the character set.

ctype ord *character*
Convert a character into its decimal numeric value. The first character of the string is converted.

ctype *?-failindex var? space string*
Tests that all characters are either a space, horizontal-tab, carriage return, newline, vertical-tab, or form-feed.

ctype *?-failindex var? print string*
Tests that all characters are a space or any character for which *ctype alnum* or *ctype punct* is true or other ‘‘printing character’’ as defined by the character set.

ctype *?-failindex var? punct string*
Tests that all characters are made up of any of the characters other than the ones for which **alnum**, **cntrl**, or **space** is true.

ctype *?-failindex var? upper string*
Tests that all characters are uppercase letters as defined by the character set.

ctype *?-failindex var? xdigit string*
Tests that all characters are valid hexadecimal digits, that is *0* through *9*, a through *f* or *A* through *F*.

If *-failindex* is specified, then the index into *string* of the first character that did not match the class is returned in *var*.

B.12.9 replicate

Syntax

```
replicate string countExpr
```

Description

Returns *string*, replicated the number of times indicated by the expression *countExpr*.

B.12.10 translit

Syntax

```
translit inrange outrange string
```

Description

Translate characters in *string*, changing characters occuring in *inrange* to the corresponding character in *outrange*. *Inrange* and *outrange* may be list of characters or a range in the form ‘A-M’. For example:

```
translit a-z A-Z foobar
```

returns "FOOBAR".

B.13 XPG/3 Message Catalog Commands

These commands provide a Tcl interface to message catalogs that are compliant with the X/Open Portability Guide, Version 3 (XPG/3).

Tcl programmers can use message catalogs to create applications that are language-independent. Through the use of message catalogs, prompts, messages, menus and so forth can exist for any number of languages, and they can altered, and new languages added, without affecting any Tcl or C source code, greatly easing the maintenance difficulties incurred by supporting multiple languages.

A default text message is passed to the command that fetches entries from message catalogs. This allows the Tcl programmer to create message catalogs containing messages in various languages, but still have a set of default messages available regardless of the presence of any message catalogs, and allow the programs to press on without difficulty when no catalogs are present.

Thus, the normal approach to using message catalogs is to ignore errors on **catopen**, in which case **catgets** will return the default message that was specified in the call.

The Tcl message catalog commands normally ignore most errors. If it is desirable to detect errors, a special option is provided. This is normally used only during debugging, to insure that message catalogs are being used. If your Unix implementation does not have XPG/3 message catalog support, stubs will be compiled in that will create a version of **catgets** that always returns the default string. This allows for easy porting of software to environments that don't have support for message catalogs.

Message catalogs are global to the process, an application with multiple Tcl interpreters within the same process may pass and share message catalog handles.

B.13.1 catclose

Syntax

```
catclose ?-fail|-nofail? cathandle
```

Description

Close the message catalog specified by *cathandle*. Normally, errors are ignored. If **-fail** is specified, any errors closing the message catalog file are returned. The **-nofail** option specifies the default behavior of not returning an error. The use of **-fail** only makes sense if it was also specified in the call to **catopen**.

B.13.2 catgets

Syntax

```
catgets catHandle setnum msgnum defaultstr
```

Description

Retrieve a message form a message catalog. *catHandle* should be a Tcl message catalog handle that was returned by **catopen**. *setnum* is the message set number, and *msgnum* is the message number. If the message catalog was not opened, or the message set or message number cannot be found, then the default string, *defaultstr*, is returned.

B.13.3 catopen

Syntax

```
catopen ?-fail|-nofail? catname
```

Description

Open the message catalog *catname*. This may be a relative path name, in which case the **NLSPATH** environment variable is searched to find an absolute path to the message catalog. A handle in the form **msgcat***N* is returned. Normally, errors are ignored, and in the case of a failed call to **catopen**, a handle is returned to an unopened message catalog. (This handle may still be passed to **catgets** and **catclose**, causing **catgets** to simply return the default string, as described above. If the **-fail** option is specified, an error is returned if the open fails. The **-nofail** option specifies the default behavior of not returning an error when **catopen** fails to open a specified message catalog. If the handle from a failed **catopen** is passed to **catgets**, the default string is returned.

B.14 Help Facility

The help facility allows you to look up help pages that were extracted from the standard Tcl manual pages and Tcl scripts during Tcl installation. Help files are structured as a multilevel tree of subjects and help pages. Help files are found by searching directories named **help** in the directories listed in the **auto_path** variable. All of the files in the list of help directories form a virtual root of the help tree. This method allows multiple applications to provide help trees without having the files reside in the same directory.

The help facility can be accessed in two ways, as interactive commands in the Extended Tcl shell or as an interactive Tk-based program (if you have built Extended Tcl with Tk).

To run the Tk-based interactive help program:

```
tclhelp ?addpaths?
```

Where *addpaths* are additional paths to search for help directories. By default, only the **auto_path** used by **tclhelp** is search. This will result in help on Tcl, Extended Tcl and Tk.

The following interactive Tcl commands and options are provided with the help package:

help	Help, without arguments, lists of all the help subjects and pages under the current help subject.
help ***subject***	Displays all of help pages and lower level subjects (if any exist) under the subject ***subject***.
help ***subject/helppage***	Display the specified help page. The help output is passed through a simple pager if output exceeds 23 lines, pausing waiting for a return to be entered. If any other character is entered, the output is terminated.
helpcd ?*subject*?	Change the current subject, which is much like the UNIX current directory. If ***subject*** is not specified, return to the top-level of the help tree. Help subject path names may also include ‘‘..’’ elements.
helppwd	Displays the current help subject.
help help | ?	Displays help on the help facility at any directory level.
apropos ***pattern***	This command locates subjects by searching their one-line descriptions for a pattern. **apropos** is useful when you can remember part of the name or description of a command, and want to search through the one-line summaries for matching lines. Full regular expressions may be specified (see the **regexp** command).

B.15 Tcl Loadable Libraries and Packages

Extended Tcl supports standard Tcl **tclIndex** libraries and package libraries. A package library file can contain multiple independent Tcl packages. A package is a named collection of related Tcl procedures and initialization code.

The package library file is just a regular Unix text file, editable with your favorite text editor, containing packages of Tcl source code. The package library file name must have the suffix **.tlib**. An index file with the suffix **.tndx**, corresponding to the package library. The **.tndx** will be automatically created by Tcl whenever it is out of date or missing (provided there is write access to the directory.

The variable **auto_path** contains a list of directories that are searched for libraries. The first time an unknown command trap is take, the indexes for the libraries are loaded into memory. If the **auto_path** variable is changed during execution of a program, it will be re-searched. Only the first package of a given name found during the execution of a program is loaded. This can be overridden with **loadlibindex** command.

The start of a package is delimited by:

```
#@package: package_name proc1 ?..procN?
```

These lines must start in column one. Everything between the **#@package:** keyword and the next **#@package:** keyword or a **#@packend** keyword, or the end of the file, becomes part of the named package. The specified procedures, *proc1..procN*, are the entry points of the package. When a command named in a package specification is executed and detected as an unknown command, all code in the specified package will be sourced. This package should define all of the procedures named on the package line, define any support procedures required by the package and do any package-specific initialization. Packages declarations maybe continued on subsequent lines using standard Tcl backslash line continuations. The **#@packend** keyword is useful to make sure only the minimum required section of code is sourced. Thus for example a large comment block at the beginning of the next file won't be loaded.

> **WARNING —** The package file should contain only TCL procedures. For example, it should not contain any "source" commands. The behaviour may be unpredictable in such cases.

Care should be taken in defining *package_name*, as the first package found in the path with a given name is loaded. This can be useful in developing new version of packages installed on the system.

For example, in a package source file, the presence of the following line:

```
#@package: directory_stack pushd popd dirs
```

says that the text lines following that line in the package file up to the next *package* line or the end of the file is a package named **directory_stack** and that an attempt to execute either

pushd, *popd* or *dirs* when the routine is not already defined will cause the **directory_stack** portion of the package file to be loaded.

B.16 Package Library Management Commands

Several commands are available for building and managing package libraries. Commands that are extended versions of the standard Tcl library commands are listed here. All of the standard Tcl library management commands and variables are also supported.

B.16.1 auto_commands

Syntax

```
auto_commands ?-loaders?
```

Description

Lists the names of all known loadable procedures and commands procedures. If **-loaders** is specified, the command that will be executed to load the command will also be returned.

B.16.2 auto_load

Syntax

```
auto_load ?command?
```

Description

Attempt to load the specified ***command*** from a loadable library. loading the package containing the procedure. If the package indexes have not been loaded for all package libraries in **auto_path**, they will be loaded. Out-of-date library indexes will be rebuilt if they are writable. The procedure returns **1** if the command was sucessfully loaded, or **0** if it was not.

Duplicated package names are skipped, the first package of a given name found in the path is loaded. If the **auto_path** has changed since the last load, indexes will be reloaded (duplicate packages will not be redefined).

If *command* is not specified, the indexes will be loaded, if they have not alreay been loaded or if the **auto_path** variable has changed, but no command will be loaded.

This command overrides the standard Tcl procedure of the same name.

B.16.3 auto_load_file

Syntax

```
auto_load_file file
```

Description

Source a file, as with the **source** command, except search **auto_path** for the file.

B.16.4 auto_packages

Syntax

```
auto_packages ?-location?
```

Description

Returns a list of the names of all defined packages. If *-location* is specified, a list of pairs of package name and the **.tlib** path name, offset and length of the package within the library.

B.16.5 buildpackageindex

Syntax

```
buildpackageindex libfilelist
```

Description

Build index files for package libraries. The argument *libfilelist* is a list of package libraries. Each name must end with the suffix **.tlib**. A corresponding **.tndx** file will be built. The user must have write access to the directory containing each library.

B.16.6 convert_lib

Syntax

```
convert_lib tclIndex packagelib ?ignore?
```

Description

Convert a Ousterhout style *tclIndex* index file and associate source files into a package library **packagelib**. If **packagelib** does not have a **.tlib** extension, one will be added. Any files specified in *tclIndex* that are in the list *ignore* will be skipped. Files listed in *ignore* should just be the base file names, not full paths.

B.16.7 loadlibindex

Syntax

```
loadlibindex libfile.tlib
```

Description

Load the package library index of the library file **libfile** (which must have the suffix *.tlib*). Package library indexes along the **auto_path** are loaded automatically on the first **demand_load**; this command is provided to explicitly load libraries that are not in the path. If the index file (with a *.tndx* suffix) does not exists or is out of date, it will be rebuilt if the user has directory permissions to create it. If a package with the same name as a package in *libfile.tlib* has already been loaded, its definition will be overridden by the new package. However, if any procedure has actually been used from the previously defined package, the procedures from *libfile.tlib* will not be loaded.

This command will also load an index built by **mkindex.tcl** program supplied with standard Tcl. This file must be named "**tclIndex**".

B.16.8 searchpath

Syntax

```
searchpath path file
```

Description

Search all directories in the specified path, which is a Tcl list, for the specified file. Returns the full path name of the file, or an empty string if the requested file could not be found.

Glossary

A

AID Key — Any 3270 special program key which causes the current screen to be sent to the 3270 SUT, and causes the SUT to transmit screen data back to the client.

Aggregate — A term used to identify a portion of a Flexible Computer Interface Format (FCIF) message. An aggregate contains zero or more tag-value pairs and is contained in an FCIF section.

App-to-App — Package that allows a user to send, receive, and analyze messages to and from a SUT over an application-to-application interface or a binary synchronous printer interface.

Array — A collection of associated variable elements.

Asynchronous Terminal Interface — An interface to an operating system or application that sends and receives data in arbitrarily-sized blocks at arbitrary times.

Attributes — **1.** The values defining the characteristics of a class or a class's connection, e.g., blinking, highlighted. **2.** The sub-commands used to specify or return attribute values. **3.** A category of methods and attributes that are used to find information about the SUT's configuration characteristics.

B

Background Execution Environment — The combination of the Script Dispatchers, the Script Engine Groups, and the associated SEs.

Background Script Engine — SE process that communicates over an channel to a controlling process.

BD — See Boot Daemon

BEE — See Background Execution Environment

Binary Synchronous Communication — An IBM communications protocol that provides access to a 3270 data stream.

Boot Daemon — A platform process required by an SD that manages the SEs running on its machine.

BSC — See Binary Synchronous Communication

BSE — See Background Script Engine

C

Character Position — The manner in which screen positions are referred to. The screen can be viewed as one long string, where the indices of that string map to a position on the screen. For instance, the first value of the string has a character position of 1, which would have a corresponding row/column value of {1 1}. The maximum character position, or the last position on the screen, varies from model type to model type, as different model types have different screen sizes. Model 2's max character

position is (24 x 80) 1920, while the bigger Model 5 has a max character position of (27 x 132) 3564.

Child Script — A Tcl script that is submitted for execution by a parent script

Class — A specific area or category of functionality.

Class Command — A command that gives you control over a class or category of functions.

Clear Tag-value Database — An ASCII file containing two columns separated by spaces or tabs. The first column contains the tag, and the second column contains the values.

CLUI — See Command Line User Interface

Command Line Script Engine — A MYNAH process that accepts Tcl commands from stdin and produces results on stdout. The Command Line Script Engine (CSE) does not interface with the MYNAH System database, but does, however, open a channel.

Concatenate — To put two items together, end to end. For example, if you concatenated the strings "uvw" and "xyz", you get "uvwxyz". If you concatenate two files, the new file contains the contents of both files, presented sequentially.

Concurrency Group — The set of all scripts that run on one Script Dispatcher

Config file — The MYNAH Configuration File (named *xmyConfig*) that resides in the directory *$XMYHOME/config*.

CSE — See Command Line Script Engine

D

des — A UNIX command to encrypt or decrypt data using the Data Encryption Standard.

Domain — A type of interface provided to the System Under Test. Examples of domains are the asysnchronous terminal interface of an application, the application-to-application interface of an application, and the synchronous printer interface of an application.

Domain Connection — Any specific input/output interface to a SUT.

E

EAB — See Extended Attribute Bytes, used in 3270 to provide more information about a field, such as color attributes.

EHLLAPI — See Emulator High Level Language Application Programmatic Interface.

Elements — Components of a list or array.

Embedded Script Engines — Script Engines that graphically display the screens associated with Term3270 or TermAsync Packages. Embedded Script Engines (ESEs) offer script execution functionality through class methods. ESEs are not separate processes.

Emulator High Level Language Application Programmatic Interface— The IBM specification API for interacting with a 3270 host providing the essential functionality underneath the MYNAH 5.0 3270 Terminal domain.

Encrypted Database — Aclear database that has been encrypted using des

ESE — See Embedded Script Engine

Exception — Any event that can abort a script.

Extended Attribute Bytes — Used by the Term3270 Package to provide more information about a field, such as color attributes.

Extended Tcl — See TclX.

Extensions — Commands and procedures that expand Tcl's capabilities.

F

FCIF — Flexible Computer Interface Format. FCIF is a text format developed at Bellcore for communicating messages between processes.

FMM — See Flexible Message Manager (TraxWay-provided wrapper to Telexel IPC)

Flexible Message Manager — A TraxWay-provided wrapper to Telexel IPC. An inter-process communication module used by the MYNAH System that uses the Telexel directory daemon underneath to do the actual IPC processing.

Focusing — Selecting a MYNAH GUI element and making it ready for you to enter information.

G

Global Array — An array of elements that are available to all domains.

GUI — Graphical User Interface.

H

Handle — A reference to an instance.

hllc() — The native EHLLAPI call. EHLLAPI makes use of one function, the hllc() function, which always takes four parameters. These four parameters determine what EHLLAPI function to execute, the input parameters to that function, and, afterwards, return the output of that function's execution, should there be any. The four parameters are commonly referred to as (and passed to the hllc() function in this order): Function Number, Data String, Data String Length or Buffer Size, and Presentation Space. This design refers to the specific EHLLAPI function simply as hllc(*function_number*). For instance, the EHLLAPI function Connect Presentation Space corresponds to hllc(1).

I

Icon — An X-Window that has been closed using a window manager function.

Iconified — The state of an X-Window after it has become an icon.

Instances — Connections made to SUTs using a class command.

IPC — Inter-process Communication Telexel IPC processes

J

Job Status Container — MYNAH GUI tool used to monitor the scripts that have submitted to the BEE.

L

List — An ordered collection of elements.

Log File — A file containing a record of activity for a software product.

M

Mask — A way to identify data that will be ignored during a comparison.

Methods — Sub-commands used to perform particular actions on instances you create in Tcl.

MYNAH System — An advanced software environment that can be used in all phases of software testing to exercise and analyze mainframe, minicomputer, and workstation applications. The MYNAH System can also be used for task automation and rapid application development.

O

OA — See Operability Agent

OM — See Operability Manager

Operability Agent — A MYNAH process that manages all MYNAH required processes on a host, communicating the start, stop and status requests to individual processes and then communicating the reply back to the OM.

Operability Management — The MYNAH mechanism, consisting of the OA and OM, that lets the MYNAH Adminstrator start, stop, and get status of all of the MYNAH processes from any host.

Operability Manager — A set of commands used interact with an OA to manageall MYNAH processes. You can start or stop a process or you can determine if a process is running statis

P

Parent Script — A Tcl script that submits other scripts for execution.

-position *position* — One of the ways of specifying screen location to a 3270 Tcl command. The position is a list of two integer values, row and column in that order. Example: -position {1 1}.

Presentation space — The 3270 screen that the EHLLAPI function call will affect and/or perform its action upon.

PRINTCOM — A program that interfaces to applications on a host computer over a binary synchronous communication line, receiving and capturing the printer output sent by the applications.

Process — An executable program that is active (running).

Prt3270 — MYNAH Package that lets a user simulate PRINTCOM processes.

R

Regression Testing — The testing of a previously verified application after changes have been made to the application.

Requesting Process — A process that sends a script-execution request to the SD (this does not include an SE sending a child-script-execution request to the SD).

Resource, X-Window System — A default value that can be changed by a user. Sets of resources are commonly stored in the *~/.Xdefaults* file (i.e., the file called ".Xdefaults" in your home directory).

Root Script — A script submitted to an SD via the GUI, CLUI, CSE, or an embedded SE, but not from one of the SEs that is controlled by the SD.

Runtime — A state in which a script is being executed.

Runtime Analysis — data Analysis that occurs during the execution of a test, and provides verification that the application being tested performed as expected. An example of runtime analysis is a comparison statement in a test script.

S

Screen Definition File — A file containing a tag name table. One file exists for each screen in a user's application. The file may contain other information in addition to the tag name table. The tag name table is delimited in this file by "begin" and "end" statements.

Screen IDs File — A file containing the names of screens and other information to uniquely identify one screen from another.

Script — A file that contains one or more instructions to be performed by a domain.

Script Dispatcher — MYNAH process that provides scheduling and concurrency control for background execution of user scripts.

Script Engine — An extended Tcl interpreter that runs user scripts.

Script Engine Group — A logical set of BSEs controlled by an SD that all run on one host and run in the same mode. When a script is submitted to the BEE, it is submitted to run in a particular SE Group. It will run on one of the SEs in that group, but it doesn’t matter which SE in the group it runs on.

Script Builder — A MYNAH GUI tool used to create script code by capturing interactions with a system, importing templates and procedures, and existing script code. A Standalone Script Builder can be run independently of the rest of the MYNAH GUI.

Script Object — A MYNAH GUI object used to create and track script code.

SD — See Script Dispatcher.

SE — See Script Engine.

SNA — See Systems Network Architecture.

String — In Tcl: A set of characters that represents the current value of a variable. In some cases, strings show what will appear on the screen or in a printout.

SUT — System Under Test.

Symbol Table — User-supplied data associated with a script. Symbol tables contain symbol-value pairs; they can be read and modified by the script during execution.

Synchronous Terminal Interface — An interface to an operating system or application that sends and receives data in blocks of predefined size at regular intervals.

Systems Network Architecture — An IBM communications protocol that provides access to a 3270 data stream.

System Under Test — The system you wish to test or which contains the application you wish to automate.

T

TagDir — A directory containing Tag Name files.

Tag Table — A formatted table in a screen definition file, containing screen information for a single screen. For each user-identified screen field this table has a name for the field (called a tag name), the field's row and column location, and the number of characters in the field.

Tag Name File — A file containing tag-value pairs, used for locating items on a synchronous screen.

Tag-value Pair — A pair of items, the first being a variable, the second being the value of that variable. Tag-value pairs reside in a symbol table.

Tags — User defined lables used by the Term3270 Package for locating items on a synchronous screen.

Tcl — Tool Command Language. An interpretive programming language, implemented as a library of C procedures, developed by John Ousterhout. Tcl is the basis for the MYNAH scripting language.

TclX — Extended Tcl, flavor of Tcl used by the MYNAH System under license from NeoSoft.

Term3270 — Package that performs 3270 synchronous terminal emulation, allowing users to build scripts that simulate an interactive work session with a SUT.

TermAsync — Package that performs asynchronous terminal emulation, allowing users to build scripts that simulate an interactive work session with a SUT.

Terminal Emulation — The use of software to emulate a type of hardware terminal (e.g., vt100, 3278).

TOPCOM — A software product that provides an interface to allow an application to establish and accept Transaction Oriented Protocol (TOP) sessions with a foreign partner, to send and receive messages to and from the partner, and to terminate the sessions. TOPCOM uses X.25 or TCP/IP transport services to transport the application data messages and TOP protocol messages between the two partners.

TOP — Extension Package that lets a user simulate TOPCOM processes.

V

Variable — A user defined quantity that can assume a value.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X