61202880L1-29.3C
April 2014

ADTRAN

Configuration Guide

Tcl Scripting in AOS

This configuration guide will assist in the use of the tool command language (Tcl) scripting language as it relates to ADTRAN Operating System (AOS) products. To achieve this objective this guide presents a basic tutorial in the Tcl language with an overview of Tcl syntax and a listing of available Tcl commands along with definitions and examples of their use. Example scripts are provided and broken down into their component elements. Commands used in interfacing Tcl with AOS, with regard to uploading and running scripts and accessing AOS commands, are explained as well.

This guide consists of the following sections:

Overview

Tcl is a highly extensible and robust language used in a wide variety of applications. It is often used in graphical user interfaces (GUIs) and for testing purposes, but is most commonly used for scripted applications. It is easy to learn, platform independent, and provides for rapid development.

In AOS applications, Tcl is most commonly used for generating scripts that help automate tasks, such as network configuration and network connectivity tests. For instance, in many organizations there are a limited number of required network configurations. In order to minimize the training and time required to set these manually, a Tcl script can be used. These scripts can access AOS commands and can be entirely automated or allow for multiple configurations within a single script by requiring user input.

The Tcl scripting language also works congruently with the AOS flash provisioning feature, allowing a Tcl script to run when initially configuring the unit. This feature minimizes the need for highly trained personnel to create relatively sophisticated configurations when installing a unit in the field.

It is also possible to run Tcl scripts based off tracks. By running scripts based off the pass or fail status of a track created to monitor network conditions, a wide variety of options related to dynamic changes in unit configuration are possible.

Hardware and Software Requirements and Limitations

Tcl support was introduced in AOS version 16.0 and is currently available in all ADTRAN hardware platforms running AOS 16.0 or later.

Basic Tcl Syntax

Tcl scripts are made up of Tcl commands. Each Tcl command is separated by a new line. Commands are organized within a line by using white space as well as different types of bracket characters that help determine how the command, or string of commands, is parsed by the Tcl interpreter. Tcl commands all use the same basic format consisting of the command name followed by one or more arguments. Arguments are comprised of operators and operands. There are a fixed number of operators in Tcl that typically correspond with the same operator in the C programming language. Operands can be numerical (integer, floating point, etc.) in nature or may consist of letters or an alphanumeric string. Operands are typically provided by user input or pre-determined by the writer of the script, and may even be defined in another command within the Tcl script.

> NOTE *Tcl commands are case sensitive. For example 'Set', 'SET', and 'set' are not equivalent. All Tcl commands should be lowercase. Variable names may contain upercase letters as long as the case is consistant throughout the script.*

Variables

Variables are a critical component of the Tcl script. They are used to hold information. This information may be predetermined in the script, generated by user input, or defined in another Tcl expression. Unlike many programming languages, variables in Tcl do not need to be declared but can be created the first time they are needed using the **set** command. Variables must start with an alphanumeric character and can consist of numbers, letters, or alphanumeric strings. A variable can also contain a list that consists of information separated by whitespace, and can be manipulated by a variety of list commands. It can also contain array data that can be manipulated by the *array* command on page 28.

Commands

Commands execute the actual work within a script. Commands typically return values that can be manipulated by the script to achieve a variety of results. Commands in Tcl follow the form of the command name followed by any number of arguments. There is an extensive set of commands available in the Tcl language. The AOS version of Tcl has a number of different commands that are specific to AOS. Additionally, some traditional Tcl commands are not available in AOS. *Table 1 on page 3* lists all Tcl commands available in AOS. Refer to the *Command Reference Guide* on page 5 for a definition of each command along with its syntax and a usage example. The *Command Reference Guide* is split into two parts. The first contains information on the most commonly used Tcl commands, and the second contains all remaining commands.

Operators

Tcl has a variety of valid operators built in. Operators are used in commands to effect a change or comparison between two or more numbers, letters, or strings. These operators perform mathematical, as well as Boolean functions, from adding and subtracting values to determining the equality or inequality of a statement. Operators are powerful tools within the scripting language. See *Table 1 on page 3* for a list of many of the basic Tcl operators. This list is not all-inclusive but should provide enough functionality for all but the most advanced scripts. For those familiar with programming in other languages the majority of these operators and their functions will be quite familiar.

Table 1. Tcl Operators

Operator	Name
!	Logical Not
*	Multiply
/	Divide
%	Remainder
+	Add
-	Subtract
<	Boolean Less Than
>	Boolean Greater Than
<=	Boolean Less or Equal
>=	Boolean Greater or Equal
==	Boolean Equal
!=	Boolean Not Equal
&&	Logical And
||	Logical Or

Special Characters

In Tcl syntax, certain special characters influence the way the script is parsed. These characters dictate much of what is required to write error-free and efficient scripts. They deal primarily with the grouping and replacement (dereferencing) of variables. Several of these characters deal with other basic necessities of language syntax. See *Table 2 on page 4* for a list of Tcl special characters. Syntactic rules and usage examples of most special characters can be found in the *Command Reference Guide* on page 5 and the *Example Scripts* on page 82.

Table 2. Tcl Special Characters

Character	Name	Definition
$	Variable Substitution	Directly precedes a variable name. Accesses a saved variable within a Tcl expression by substituting the variable name with its stored value.
[ ]	Subcommand Substitution	Bracketed commands will be parsed and replaced inline with its resulting value.
“ ”	Word Grouping with Substitution	Bracketed words are handled as a single argument, but variables will be substituted with their stored value.
{ }	Word Grouping without Substitution	Braced words are handled as a single argument, but variables will not be replaced.
\	Backslash Substitution	Any subsequent character is replaced by itself. This escape mechanism allows other special characters to be used without triggering special processing. When placed at the end of a line of code, it allows the code to continue on to the next line. Good for long strings of code.
#	Comment	Appearing at the beginning of a statement any text or commands that follow will not be parsed and will return no result.
;	Statement Separator	Serves as a separator that allows a new script to follow immediately as if it was placed on a new line.

NOTE *Examples for the use of special characters are contained within the usage examples in the* ***Command Reference Guide*** *on page 5*

Command Reference Guide

All Tcl commands available in AOS are defined in the table below. They are listed in two categories: **Commonly Used Commands** and **Additional Commands**.

NOTE *To avoid confusion, variables within command syntax descriptions have been enclosed in parentheses. Different types and positions of bracketing can be used within a command depending on the desired results. See Table 2 on page 4 for a description of bracketing types and functions.*

Table 3. Tcl Command List

Commonly Used Commands		Additional Commands			
Page		Page		Page	
6	cli	26	add64	54	incr
7	dir	27	append	55	info
8	echo	28	array	57	join
9	eval	29	break	58	lappend
10	expr	30	case	59	linsert
11	fileinfo	31	catch	60	list
12	for	32	cmdtrace	61	lrange
13	foreach	33	concat	62	lreplace
14	if	34	continue	63	lsearch
15	input	35	copy	64	lsort
16	isset	36	currenttime	65	proc
17	lindex	37	delete	66	regexp
18	llength	38	error	68	regsub
19	read	39	event	69	rename
20	set	40	format	70	return
21	sleep	41	formattime	71	scan
22	string	43	geticmpmessages	73	settime
24	while	45	getportname	74	split
25	write	46	getprotocolname	75	strtoip
		47	getprotocols	76	subtract64
		48	gettcpports	77	trace
		50	getudpports	78	unset
		52	global	79	uplevel
		53	greaterthan64	80	upvar

cli

The **cli** command is one of the most important AOS Tcl commands because it allows you to access any AOS command. This gives your scripts the the ability to automate routine tasks within the system. This command is issued in the Global Configuration mode. In order to issue a command in the Enable level the AOS **do** command is added before the AOS command.

Syntax Description

cli [do] *(AOS command)*

Usage Examples

The following example prints the results of the **cli** command, that tells the unit to run the AOS **traceroute** command, to the screen.

```
echo “Performing a traceroute:
[cli do traceroute 192.168.0.1]”
```

Output

```
Performing a traceroute:
Type CTRL+C to abort
Tracing route to 192.168.0.1 over a maximum of 30 hops
122ms20ms20ms192.168.0.65
223ms20ms20ms192.168.0.1
```

dir

The **dir** command returns a list of all files (without paths) located in the specified directory. If the desired directory is located in a different directory than where the script is run then a full path must be provided.

Syntax Description

dir *(directory name)*

Usage Examples

The following example prints to the screen a list of all files in the root directory.

echo “[dir //]”

Output

```
3200start
NV3200A-16-00-18-E.biz
NV3200B-boot-11-02-05.biz
startup-config
startup-config.bak
test1.tcl
```

echo

The **echo** command prints any argument to the screen. Typically the double quotes symbol is used to indicate a grouping of words to be printed to the screen. The double quotes symbol preserves the spacing of the characters within. However, in some cases no grouping is used and other times brackets might be required to create the desired output.

Syntax Description

echo *(arguments)*

Usage Examples

The following example prints the specified argument to the screen.

echo "This file is a test."

Output

This file is a test.

eval

The **eval** command forces the evaluation of one or more arguments. It concatenates all arguments and passes them to the Tcl interpreter recursively. Rather than making a single pass, the Tcl interpreter will parse multiple times until the script is evaluated to its logical conclusion. The result of the evaluation is the returned value, or any error message that is generated during the evaluation. This command is useful for storing scripts within variables to be evaluated at a later time.

Syntax Description

eval *(argument)*

Usage Examples

The following example prints to the screen the result of the evaluation of the specified argument.

```
Set myvariable “set v 100”
Eval $myvariable
Echo $v
```

Output:

```
100
```

expr

The **expr** command concatenates all arguments and evaluates the result as a Tcl expression. Expressions are composed of math functions and/or operators that are combined with arguments (operands) to create valid Tcl expressions, and generally yield numeric results. For more information on operators used in Tcl expressions, see Table 1 on page 3.

Syntax Description

expr *(arguments)*

Usage Examples

The following example prints to the screen the result of the mathematical expression.

```
set x “expr {20 / 4}”
echo $x
```

Output:

```
5
```

fileinfo

The fileinfo command is one of several AOS-specific Tcl commands that have been included to simplify file management. By using one of several options this command allows you to obtain specific information about the given file, directory, or device. Variations of this command include:

fileinfo size <filename> returns the size of the given file in bytes

fileinfo date <filename> returns the date of the given files creation. This is returned as seconds since January 1st 1970, 12:00 AM.

fileinfo exists <filename> returns a 1 if the file or directory exists and a 0 if it does not.

fileinfo isdir <name> returns a 1 if the object in question is a directory and 0 if it is a file.

fileinfo free <device name> returns the free space available on the device.

fileinfo total <device name> returns the total space available on the device.

fileinfo used <device name> returns the total space used on the device.

Syntax Description

fileinfo size *(filename)*

fileinfo date *(filename)*

fileinfo exists *(filename)*

fileinfo isdir *(name)*

fileinfo free *(device name)*

fileinfo total *(device name)*

fileinfo used *(device name)*

Usage Examples

The following example tests for the existence of a file named file1and prints the results of that test to the screen.

```
write file1 “This is a test file.”
set test  [fileinfo exists file1]
if {$test == 1} {
     echo “Valid Tcl file.”
} else {
     echo “Not a valid Tcl file.”
}
```

Output

Valid Tcl file.

for

The **for** command is a looping command. The *(start)*, *(next)*, and *(body)* arguments must be Tcl command strings. The *(test)* argument should be a Tcl expression string. The *(start)* string is executed first. The *(test)* string is then evaluated as a Tcl expression. If the result of *(test)* is not zero, then the *(body)* argument is executed followed by the (next) argument. The command continues to loop in this manner until *(test)* finally equals *0*.

Syntax Description

for *(start) (test) (next) (body)*

Usage Examples

The following example evaluates the *(test)* argument and continues to increment the variable as indicated in the *(body)* argument until the *(test)* argument no longer passes.

```
for {set x 0} {$x<10} {incr x} {
    echo “x is $x”
}
```

Output

```
x is 0
x is 1
x is 2
x is 3
x is 4
x is 5
x is 6
x is 7
x is 8
x is 9
```

foreach

The **foreach** command is a looping command. For each element in *(list)*, the command assigns that element (in order from first to last) to *(variable name)*. The *(body)* argument is a Tcl command string that is then executed for each element in *(list)*. The total number of loops is determined by the number of elements in *(list)*. The **in** argument is optional and exists for clarity, having no effect on the command.

> *The output below is identical to the output of the for command. In the case of the for command, the* ***incr*** *command is necessary in order to achieve this output.*

Syntax Description

foreach *(variable name)* **in** *(list) (body)*

Usage Examples

The following example prints to the screen the value of the variable, as indicated by the body argument, for each value provided in the list.

```
foreach x {1 2 3 4 5 6 7 8 9} {
    echo “x is $x”
}
```

Output

```
x is 1
x is 2
x is 3
x is 4
x is 5
x is 6
x is 7
x is 8
x is 9
```

if

The **if** command concatenates and evaluates its arguments as Tcl expressions. Expressions can be Boolean values where **0** is false and anything else is true or they can be a string value such as **true** or **yes** for true and **false** or **no** for false. If the expression is true, then the body is executed and at this point the command is finished. Otherwise, the next **elseif** expression is evaluated and, if true, its body executed, and so on. If none of the expressions are true, then the final **else** body (*body4* in the example) is executed.

There can be any number of **elseif**/**then** arguments included in an **if** command. The **then** and **elseif** argments are not required and are used primarily for formatting and legibility of the code. If no final **else** argument is used at the end, then the final body statement will be invalid. If none of the previous expressions matched, then the value returned would be an empty string. Otherwise, the command returns the value of the body script that was executed.

Syntax Description

if *(expression1)* **then** (*body1)* **elseif** *(expression2)* **then** *(body2)* **elseif** *(expression3)* **then** (*body3)* **else** *(body4)*

Usage Examples

The following example checks the value of the variable against several options and then performs the body argument corresponding to the matching expression.

```
set variable 2

if {$variable == 0} {
    echo "the answer is 0"
} elseif {$variable == 1} {
    echo "the answer is 1"
} elseif {$variable == 2} {
    echo "the answer is 2"
} else {
    echo "variable is not 0, 1, or 2"
}
```

Output

```
the answer is 2
```

input

The **input** command is used when a script requires user input. The **input** command pauses the script and waits for user-returned input. The command returns what the user typed in after a carriage return. The returned value can be stored in a variable and used by other commands within the script. No arguments are necessary when using the **input** command. This is an AOS specific command.

Syntax Description

```
[input]
```

Usage Examples

The following example outputs the echoed text to the screen and uses the **input** command to pause the script and wait for user input.

```
echo “Are you right or left handed? (R/L\)”
set variable [input]
echo “You entered $variable”
```

Output

```
Are you right or left handed?
You entered L
```

isset

The **isset** command tests for the existence of a given variable. If the variable is set the command returns a **1**. If the variable is not set the command returns a **0**.

Syntax Description

isset *(variable name)*

Usage Examples

The following example checks for the existence of the variable and prints the results to the screen.

```
set variable 17

echo “[isset {variable}]”
```

Output

```
1
```

lindex

The **lindex** command is designed to retrieve a specified element from a provided list. Elements are numbered beginning with 0. If the index number is greater than or equal to the number of elements in the list, then an empty string is returned. If instead of an integer, the argument **end** is provided in the place of **index** then the final element of **list** is returned.

Syntax Description

lindex *(list) (index)*

lindex *(list)* **end**

Usage Examples

The following example retrieves the fifth element, based on the index of four, from the provided list and prints the result to the screen.

```
echo "[lindex {1 2 3 4 5 6 7 8} 4]"
```

Output

```
5
```

llength

The **llength** command returns a number equal to the number of elements in the provided list.

Syntax Description

llength *(list)*

Usage Examples

The following example prints to the screen the total number of elements in the provided list.

```
echo "[llength {a b c d e f g h I j}]"
```

Output

```
10
```

read

The **read** command is often used in conjunction with the **write** (refer to page 25) command. The contents of the *(filename)* will be returned. It can be used to return the contents of another Tcl script file, which can then be executed, if a Tcl script has been broken down into multiple files for easier maintenance.

Syntax Description:

read *(filename)*

Usage Examples

The following example creates a user-defined procedure to read the contents of a file that is then evaluated by the Tcl interpreter.

```
write testfile.tcl [echo "This file is a test."]
rename testfile.tcl testfile2.tcl

proc returnfile {filename} {
    eval [read $filename]
}

returnfile testfile2.tcl
```

Output

```
This file is a test.
```

set

The **set** command is used to create variables and give them a value. *(name)* is the name of the variable and *(value)* is the string to be stored in the variable. The set command can also be used to return the value stored in an existing variable by executing the command without the *(value)* argument.

> NOTE: If the variable name is followed by an element within parenthesis it is indicative of an array element. The enclosed element is the index of the array and the name before the parenthesis is the array name.

Syntax Description

set *(name) (value)*

set *(name)*

Usage Examples

The following example sets the value of a variable and prints that value to the screen.

```
set variable 10


echo “$variable”
```

Output

```
10
```

sleep

The **sleep** command is used to pause a script at a desired point. The *(seconds)* argument establishes the length of the pause.

Syntax Description:

sleep *(seconds)*

Usage Examples:

The following example pauses the script for 15 seconds before continuing to evaluate the remaining contents of the script.

```
sleep 15
```

string

The **string** command is quite flexible with a number of available option arguments. This option argument dictates the type of string operation performed. Common operations include string comparisons, finding the length of the string, finding characters within the string, replacing characters within the string, and returning characters from within the string.

string compare performs a character by character comparison between two strings. The command returns **-1**, **0**, or **1** depending on whether *(string1)* is alphabetically less than, equal to, or greater than *(string2)*.

string first searches *(string2)* for a sequence of characters that matches exactly the characters contained in *(string1)*. If a match is found the command returns the index of the first character in the first match. If a match is not found -1 is returned.

string index returns the character located within the string that corresponds to the number located in the *(character index)* argument. Characters are counted beginning with 0. If the *(character index)* is greater than or equal to the number of characters within the string then an empty string is returned.

string last searches *(string2)* for a sequence of characters that matches exactly the characters contained within *(string1)*. If a match is found the command returns the index of the first character in the last match. If a match is not found -1 is returned.

string length Returns a decimal string corresponding to the number of characters within the string.

string match compares a pattern to the provided *(string)*. If the pattern matches the *(string)* contents exactly a 1 is returned. If it does not match a 0 is returned. Within the pattern certain special sequences may appear.

> ***** Matches any sequence of characters in the *(string)*.
>
> **?** Matches any single character in the *(string)*.
>
> [**chars**] Matches any character in the set given by chars. If a sequence of the form x-y appears in chars, any character between x and y will match.
>
> \x Matches the single character x, where x is one of the following: * ? [ ] \\. This prevents the interpretation of the characters * ? [ ] \\ in a pattern.

Syntax Description

string compare *(string1) (string2)*

string first *(string1) (string2)*

string index *(string) (character index)*

string last *(string1) (string2)*

string length *(string)*

string match *(pattern) (string)*

Usage Examples

The following example compares the user input with provided string and prints the appropriate result to the screen.

```
echo “Would you like to test a string match?”
set answer [input]
if {[string match “[yY][eE][sS]”}{
    echo “You typed yes”
}else{
    echo “You didn’t type yes, but we tested it anyway.”
}
```

Output

```
Would you like to test a string match?
you typed yes
```

while

The **while** command repeats the *(body)* continually for as long as the *(test)* condition is met. The *(test)* condition is evaluated and, if the Boolean result is true, the *(body)* is executed. The *(test)* condition is evaluated again and the process repeats until the test condition returns a false value.

Syntax Description

while *(test) (body)*

Usage Examples

The following example sets the variable **x** to **5**. Then, using the while command, the *(test)* argument is evaluated and the value of the variable is displayed. The remainder of the *(body script)* increments the value of the variable and continues to do so as long as the *(test)* argument passes.

```
set x 5

while {$x < 10} {
  echo "x is $x"
  incr x
}
```

Output

```
x is 5
x is 6
x is 7
x is 8
x is 9
```

write

The **write** command writes the given *(string)* to a file with the specified *(filename)*. The string is returned using the **read** command (refer to page 19).

Syntax Description

write *(filename) (string)*

Usage Examples

The following example uses the **write** command to create a file named **testfile**. The contents of **testfile** are then printed to the screen.

```
write testfile “This file is a test.”


echo "[read testfile]"
```

Output

```
This file is a test.
```

add64

The **add64** command adds two number arguments and returns the total value of the numbers. The **add64** command allows 24-digit (64-bit) numbers where the **expr** (refer to page 10) command only allows 10-digit signed (32-bit) numbers.

Syntax Description

Add64 *(argument 1) (argument 2)*

Usage Examples

The following example adds two arguments together and prints the results to the screen.

```
echo "[add64 12398745879 27567839423]"
```

Output

```
39966585302
```

append

The append command is used when necessary to add values to the end of an existing set variable. The *(variable name)* is accessed and the data contained in the *(value)* argument is appended to the existing values stored in the variable.

Syntax Description

append *(variable name) (value)*

Usage Examples

The following example sets a variable, appends new data to the variable, and prints the results to the screen.

```
set  variable “a b c ”


append variable “d e”


echo $variable
```

Output

```
a b c d e
```

array

The array command allows for the manipulation of data arrays. An **array** is a searchable set of indexed data. Each piece of data within the array is paired with an element name or index. Several **array** options are available for the manipulation and reading of the data contained within the specified array. These include setting array values, searching for specific data within an array, and returning the data stored within the array. Options for this command include:

array anymore Determines if all elements from the specified *(search ID)* have been processed. If a 1 is returned there are still unprocessed elements remaining. If all elements have been processed a 0 is returned.

array donesearch Permanently terminate an array search for the specified *(search ID)*.

array names Returns a list of indexes for the array. If a *(pattern)* is specified the command only returns the indexes that match the pattern.

array nextelement Returns the next element index within a given array that has not yet been processed. If all elements have been processed an empty string is returned.

array size Returns a decimal string equal to the number of elements in the array. It returns 0 if the *(array name)* is not valid.

array startsearch Initiates an element by element search of a given array. The command returns a *search ID* value.

Syntax Description

array anymore *(array name) (search ID)*

array donesearch *(array name) (search ID)*

array names *(array name) (pattern)*

array nextelement *(array name) (search ID)*

array size *(array name)*

array startsearch *(array name)*

Usage Examples

The following example creates an array named **FavoriteColors** that contains two value/index pairs.The array is then searched and its contents formatted and printed to the screen.

```
set FavoriteColors(jack) blue
set FavoriteColors(jill) green
set id [array startsearch FavoriteColors]
while {[set name [array nextelement FavoriteColors $id]] != ""} {
    echo "$name likes $FavoriteColors($name)"
}
array donesearch FavoriteColors $id
```

Output

```
jack likes blue
jill likes green
```

break

The **break** command causes the termination of a loop within a looping command. When located in nested loops, the **break** command causes the script to abort the innermost looping command. No arguments are necessary.

Syntax Description

```
break
```

Usage Examples

The following example uses the **break** command to terminate a loop once the test conditions are met.

```
set variable 0
while 1 {
  echo $variable
  incr variable
  if {$variable == 10} break
}
```

Output

```
0
1
2
3
4
5
6
7
8
9
```

case

The **case** command matches the value of *(string)* against each *(pattern list)*. If *(string)* matches a *(pattern list)* then the following *(body)* argument is executed by the Tcl interpreter. Wildcards may be used within a *(pattern list)*. Refer to **string match** under the **string** command on page page 22 for further information on wildcards. This command is similar in function to the **switch** command found in some versions of Tcl.

Syntax Description

case *(string) (pattern list) (body) (pattern list) (body)*

Usage Examples

The following example sets the value of a string and uses the case command to match a string to one of two possible options. The results of that match are then executed, printing the appropriate text to the screen.

```
set string def
case $string {abc} {echo "match1"} {def} {echo "match2"}
```

Output

```
match2
```

catch

The **catch** command prevents a script from halting when an error occurs. Typically a Tcl script aborts when an error occurs. Using the **catch** command the specified error message is displayed and the specified *(script)* continues to run allowing the script to handle errors internally. If an error occurs within the script a **1** is set as the value of *(variable name)*. If no error occurs then a **0** is set as the value of *(variable name)*.

Syntax Description

catch *(script) (variable name)*

Usage Examples

The following example uses the **catch** command to print an error message to the screen while allowing the script to continue running.

```
catch {set variable2 [open "invalidfile"]} variable

if {$variable == 1} {
   echo "An error occurred"
}
```

Output

```
An error occurred
```

NOTE *This output is returned only if **invalidfile** is actually invalid*

cmdtrace

The **cmdtrace** command prints a trace statement for all commands executed at the specified level or below. The top *(level)* is indicated by a 1. If *(on)* is specified all commands at all levels are traced. The AOS **debug tcl** command must be enabled for **cmdtrace** messages to be printed to the screen. This command is primarily used during script writing, to aid in troubleshooting. Multiple command options may be used to achieve the desired trace output. Options for this command include:

notruncate Prevents the truncation of all commands and evaluated arguments.

noeval Causes arguments to be printed unevaluated.

flush Causes a stdio flush to occur after every line is written.

procs Allows only procedure calls to be traced.

Syntax Description

cmdtrace *(on / level)*

cmdtrace *(on / level)* **notruncate**

cmdtrace *(on / level)* **noeval**

cmdtrace *(on / level)* **flush**

cmdtrace *(on / level)* **procs**

Usage Examples

The following example uses a script containing a **for** loop. The **cmdtrace** command is used to print the scripts debugging information to the screen.

```
cmdtrace on
for {set i 0} {$i < 3} {incr i} {
    echo [expr {$i - 1}]
}
cmdtrace off
```

Output

```
2007.07.11 9>30:50 TCL_CLI.test1.tcl script starting
2007.07.11 9>30:50 TCL_CLI.test1.tcl 1: for {set i 0} {$1 , 4} {incr i} {
\n        echo [expr {$i - 1}]\n}
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: set i 0
2007.07.11 9>30:50 TCL_CLI.test1.tcl 3: expr {$i - 1}
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: echo -1
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: incr i
2007.07.11 9>30:50 TCL_CLI.test1.tcl 3: expr {$i - 1}
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: echo 0
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: incr i
2007.07.11 9>30:50 TCL_CLI.test1.tcl 3: expr {$i - 1}
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: echo 1
2007.07.11 9>30:50 TCL_CLI.test1.tcl 2: incr i
2007.07.11 9>30:50 TCL_CLI.test1.tcl 1: cmdtrace off -1
0
1
```

concat

The **concat** command combines any number of lists into one list. It removes any excess whitespace from the leading and trailing list elements, and leaves a single space between each list argument. The command returns the concatenated list.

Syntax Description

concat *(list1) (list2) (list3)*

Usage Examples

The following example concatenates a list so that excess characters and whitespace are removed.

```
echo [concat a b {c d e} {f {g h}}]
```

Output

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

continue

Like the **break** (refer to page 29) command, the **continue** command is typically called within a looping command. It forces the current script to abort the innermost looping command, but, unlike break, it continues forward with the next iteration of the loop. No argument is required for the **continue** command.

Syntax Description

continue

Usage Examples

In the following example the **continue** command is used to skip the body argument for several iterations of a looping command.

```
foreach x {0 1 2 3 4 5 6 7 8 9} {
if {$x < 3} continue
    echo $x
}
```

Output

```
3
4
5
6
7
8
9
```

copy

The **copy** command creates a duplicate of a specified file and saves it to a new filename.

Syntax Description

copy *(filename) (new filename)*

Usage Examples

The following example copies the contents of testfile.tcl to testfile2.tcl and returns the contents of the second file to the screen using a user defined procedure.

```
write testfile.tcl [echo "This file is a test."]

copy testfile.tcl testfile2.tcl

proc returnfile {filename} {
    eval [read $filename]
}

returnfile testfile2.tcl
```

Output

```
This file is a test.
```

currenttime

The **currenttime** command sets a valid Tcl variable with a value that corresponds to the time, in seconds, since January 1st, 1970, 12:00 AM.

Syntax Description

currenttime *(variable)*

Usage Examples

The following example sets the current time and prints it to the screen along with a legibly formatted version of the current time (Refer to *formattime* on page 41).

```
set time 0
currenttime time
echo "$time is the number of seconds since January 1, 1970"
set formattedtime 0
formattime $time formattedtime "EEE, MMM d, yyyy HH:mm:ss zzzz"
echo "the time is: $formattedtime"
```

Output

```
1173567451 is the number of seconds since January 1, 1970
the time is: Sat, Mar 10, 2007 22:57:31 Universal Coordinated Time
```

delete

The delete command removes a specified file from the unit.

Syntax Description

delete *(filename)*

Usage Examples

The following example uses the **fileinfo** command to test the existence of a file that was deleted using the **delete** command and prints the results to the screen.

```
write file1 "This is a test file."
delete file1
if {[fileinfo exists $file1]}{
     echo “Valid file.”
}else{
     echo “Not a valid file.”
}set test  [fileinfo exists file1]
```

Output

```
Not a valid file.
```

error

Use the **error** command to exit the script immediately with the specified error message.

Syntax Description

error *(error message) (error info) (error code)*

Usage Examples

The following example defines an error message to be printed to the screen if the variable value is not numeric.

```
set variable a
if ![regexp {^[0-9]+$} $variable] {
 error "variable must be numeric"
}
echo “you should not see this”
```

Output

```
variable must be numeric
```

event

The **event** command creates custom event messages that are logged through the AOS Event System and, when configured, through SYSLOG. The event command *(type)* is a priority level that affects how the AOS Event System handles the user-defined *(message text)*. These levels are, from lowest to highest priority, **debug**, **info**, **notice**, **warning**, **error**, and **fatal**.

Syntax Description

event debug *(message text)*

event info *(message text)*

event notice *(message text)*

event warning *(message text)*

event error *(message text)*

event fatal *(message text)*

Usage Examples

The following example creates a notice level event along with a custom message to indicate when a particular event that the script is monitoring has occurred. This event can be forwarded to a SYSLOG server if available.

event notice “ppp 1 went down, changing configuration to use ppp 3 for DSCP tagging”

format

The **format** command uses a *(format string)* as a pattern for output. Using the % symbol and the letter of the desired conversion option (similar to the ANSI C sprintf procedure), the values provided in each *(argument)* are converted and subsituted into the resulting output. There must be one conversion option for each *(argument).*

%d Converts integer to a signed decimal string.

%u Converts integer to an unsigned decimal string.

%i Converts integer to a signed decimal string. The integer may either be in decimal, in octal (with leading 0) or in hexadecimal (with leading 0x).

%o Converts integer to an unsigned octal string.

%x Converts integer to unsigned hexadecimal string, using digits.

%c Converts integer to the Unicode character it represents.

%s No conversion; just insert string.

%f Converts floating-point number to a signed decimal string (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 Converts floating-point number to scientific notation in the form x.yyye±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.

%g If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for **%e**. Otherwise, it is converted as for **%f**. Trailing zeroes and trailing decimal point are omitted.

Syntax Description

format *(format string) (argument1) (argument2) (argument3)...*

Usage Examples

The following example converts the two variables to their hexadecimal equivalent and prints the results to the screen.

```
set variable 10
set variable2 18
echo "[format "%x %x" $variable $variable2]"
```

Output

```
a 12
```

formattime

The **formattime** command is used to display time in a user-friendly format. The *(input variable)* time is a measure of time based on the number of seconds since January 1,1970, 12:00 AM. This value can be read by using the **currenttime** command (refer to page 36) and established using the **settime** command (refer to page 73). The time in *(input variable)* is accessed, formatted using the pattern contained in *(format)*, and stored in the *(output variable)*. The number and type of characters (shown in the table below) contained in the *(format)* portion affect how the time stored in the *(output variable)* is displayed. If a character is used four or more times back to back, then the full formatted display is used, whereas if fewer than four characters are used, the display will be abbreviated if an abbreviated version is available. For example, EEEEEEE displays Tuesday, where EEE displays Tue.

Symbol	Definition	Example
G	Era Designator	AD
y	Year	2007
M	Month	6-June
d	Day of Month	30
h	Hour in a 12 Hour Clock (1-12)	10
H	Hour, 0-23	21
m	Minute	37
s	Second	17
S	Millisecond	852
E	Day of Week	Tuesday
D	Number of Day in Year	181
a/p	AM/PM	PM
k	Hour, 1-24	24
K	Hour in a 12 hour clock (0-11)	0
z	Time Zone	CST
‘	Escape for Text	
‘ ‘	Single Quote	‘

Syntax Description

formattime *(input variable) (output variable) (format)*

Usage Examples

The following example sets the current time and prints it to the screen along with a legibly formatted version of the current time.

```
set mytime 0
set output 0
settime mytim2 2007 5 1 8 27 30
echo "$mytime"
formattime $mytime output "EEE, MMM d, yyyy HH:mm:ss z"
echo "the time is: $output"
```

Output

```
1178008050
the time is: Tue, May 1, 2007 08:27:30 UTC
```

geticmpmessages

The **geticmpmessages** command returns a numeric list of Internet Control Message Protocol (ICMP) codes along with their corresponding types and messages. These values are saved as an array with the specified *(variable name)*. The format for the array is the ICMP code followed by its type then its name.

Syntax Description

geticmpmessages *(variable name)*

Usage Examples

The following example returns an array of all ICMP information and formats that information before printing it to the screen.

```
geticmpmessages messages
set length [array size messages]
echo "name              code  type"
for {set i 0} {$i < $length} {incr i} {
        echo "[lindex $messages($i) 2]    [lindex $messages($i) 0]    [lindex $messages($i) 1]"
}
```

Output

```
name code type
echo-reply0 0
precedence-unreachable 1 32783
net-unreachable    3  0
host-unreachable    3   1
protocol-unreachable    3        2
port-unreachable    3   3
packet-too-big    3      4
source-route-failed    3        5
network-unknown    3     6
host-unknown    3     7
host-isolated    3     8
dod-net-prohibited    3     9
dod-host-prohibited    3     10
net-tos-unreachable    3     11
host-tos-unreachable    3     12
administratively-prohibited    3    13
unreachable    3     65535
source-quench    4     0
net-redirect    5     0
host-redirect    5     1
net-tos-redirect    5     2
```

host-tos-redirect	5	3
redirect	5	65535
alternate-address	6	0
echo	8	0
router-advertisement	9	0
router-solicitation	10	0
ttl-exceeded	11	0
reassembly-timeout	11	1
option-missing	12	1
timestamp-request	13	0
timestamp-reply	14	0
information-request	15	0
information-reply	16	0
mask-request	17	0
mask-reply	18	0
traceroute	30	65535
conversion-error	31	65535
mobile-redirect	32	65535

getportname

The **getportname** command returns a well-known port number and name based on the specified *(protocol)* (TCP or UDP) and the *(port number)*. The command returns a list with the specified port number followed by the port name. If a *(port number)* and *(protocol)* do not have a well-known port name then an empty string is returned.

Syntax Description

getportname *(protocol) (port number)*

Usage Examples

The following example returns the number and name of the specified port to the screen.

```
echo "[getportname tcp 80]"
```

Output

```
80 www
```

getprotocolname

The **getprotocolname** command returns the well-known port number and name of the IANA protocol associated with the specified *(number)*. The command returns a list with the number followed by the name. If the specified *(number)* is not associated with a valid protocol, then an empty string is returned.

Syntax Description

getprotocolname *(number)*

Usage Examples

The following example returns the name and number of the desired protocol and prints them to the screen.

```
echo "[getprotocolname 6]"
```

Output

```
6 tcp
```

getprotocols

The **getprotocols** command returns a list of IANA protocol numbers and well known names. This command creates an array called *(variable name)* which is an array of lists where the list format is the protocol number followed by its name.

Syntax Description

getprotocols *(variable name)*

Usage Examples

The following example returns an array of all protocol information and formats that information before printing it to the screen.

```
getprotocols protocols
set length [array size protocols]
echo "number name"
for {set i 0} {$i < $length} {incr i} {
        echo "[lindex $protocols($i) 0]    [lindex $protocols($i) 1]"
}
```

Output

```
number name
1 icmp
4 ip
6 tcp
17 udp
47 gre
50 esp
51 ahp
```

gettcpports

The **gettcpports** command returns a numeric list of Transmission Control Protocol (TCP) well-known port numbers and names. This command creates an array called *(variable name)* which is an array of lists where the list format is the protocol number followed by its name.

Syntax Description

gettcpports *(variable name)*

Usage Examples

The following example returns an array of all TCP port information and formats that information before printing it to the screen.

```
gettcpports ports
set length [array size ports]
echo "num name"
for {set i 0} {$i < $length} {incr i} {
        echo "[lindex $ports($i) 0]    [lindex $ports($i) 1]"
}
```

Output

```
num name
7 echo
9 discard
13 daytime
19 chargen
20 ftp-data
21 ftp
22 ssh
23 telnet
25 smtp
37 time
43 whois
49 tacacs
53 domain
70 gopher
79 finger
80 www
101 hostname
109 pop2
110 pop3
111 sunrpc
113 ident
```

119 nntp

179 bgp

194 irc

443 https

496 pim-auto-rp

512 exec

513 login

514 syslog

515 lpd

517 talk

540 uucp

543 klogin

544 kshell

getudpports

The **getudpports** command returns a numeric list of User Datagram Protocol (UDP) well-known port numbers and names. This command creates an array called *(variable name)* which is an array of lists where the list format is the protocol number followed by its name.

Syntax Description

getudpports *(variable name)*

Usage Examples

The following example returns an array of all UDP port information and formats that information before printing it to the screen.

```
getudpports ports
        set length [array size ports]
        echo "num name"
        for {set i 0} {$i < $length} {incr i} {
        echo "[lindex $ports($i) 0]    [lindex $ports($i) 1]"
        }
```

Output

```
num name
7 echo
9 discard
37 time
42 nameserver
49 tacacs
53 domain
67 bootps
68 bootpc
69 tft
111 sunrpc
123 ntp
137 netbios-ns
138 netbios-dgm
139 netbios-ss
161 snmp
162 snmptrap
177 xdmcp
195 dnsix
434 mobile-ip
496 pim-auto-rp
```

500 isakmp

512 biff

513 who

514 syslog

517 talk

520 rip

global

The **global** command allows the writer of the script to access global variables within a **proc** procedure body, rather than just local variables defined within the procedure. The procedure is essentially a user defined command that can be called within the script.

Syntax Description

global *(variable name)*

Usage Examples

The following example accesses the variable created on the first line of the script within the user defined procedure.

```
set globalvar 5
proc sum {} {
    set variable2 10
    global globalvar
    set answer [expr {$globalvar + $variable2}]
    return $answer
}
echo "[sum]"
```

Output

```
15
```

greaterthan64

The **greaterthan64** command evaluates whether or not the value of the first argument is greater than the value of the second argument. If the value is greater, a 1 is returned. If the value is less, a 0 is returned. The **greaterthan64** command allows 24-digit (64-bit) unsigned numbers where the **expr** (refer to page 10) command only allows 10-digit (32-bit) signed numbers.

Syntax Description

greaterthan64 *(argument 1) (argument 2)*

Usage Examples

The following example compares the value of a variable to 10 to see if it is greater, and displays the result to the screen.

```
set variable 17

if [greaterthan64 $variable 10] then {
    echo "$variable is greater than 10"
} else {
    echo "$variable is less than or equal to 10"
}
```

Output

```
17 is greater than 10
```

incr

The **incr** command takes an existing integer *(variable name)* and increments it. By default the *(variable name)* is incremented by 1, but specifying an *(increment)* value after the *(variable name)* will cause the number to increment by that value instead, including -1.

Syntax Description

incr *(variable name) (increment)*

Usage Examples

The following example increments the value of a variable by 10 and returns that result to the screen.

```
set mynumber 15
set variable [incr mynumber 10]
echo "$variable"
```

Output

```
25
```

info

The **info** command is intended to return a variety of information concerning the current state of the Tcl interpreter depending on which option is chosen from the available possibilities.

info args returns a list containing the names of all arguments located within the specified Tcl procedure.

info body returns the entire body contents of the specified procedure.

info cmdcount returns the total number of commands parsed by the Tcl interpreter.

info commands returns a list of all available Tcl commands including user defined procedures. If a pattern string is specified only commands matching that pattern will be returned.

info complete returns a 1 if the specified command seems to have been written in a syntactically complete manner. If this is not the case then a 0 is returned.

info default returns the default value of an argument within a user defined procedure and stores it in a variable. If the argument has no default value then 0 is returned.

info exists returns a 1 if the indicated variable name currently exists and has a set value either globally or within a user defined procedure. If the variable does not currently exist a 0 is returned.

info globals returns a list of all currently defined global variables. If a pattern string is specified only variables matching that pattern will be returned.

info level returns a number equal to the stack level of the command calling for the information. If a number string is specified then the command name and arguments are returned of the command that is equal to that command level in the stack.

info library returns the value of the tcl_library variable, which contains the location of the directory where the Tcl variable library is stored.

info locals returns a list of all local variables defined within the current procedure. Global variables used in the script will not be returned. If a pattern string is specified only variables matching the pattern will be returned.

info procs returns a named list of all user defined procedures. If a pattern string is specified only procedures matching that pattern will be returned.

info script returns the innermost file name of the script that is currently being processed by the Tcl interpreter.

info tclversion returns the value of the global tcl_version variable.

info vars returns a list of all visible variables. A visible variable includes both local and global variables as well as variables which do not currently have set values but have been established by the variable command. If a pattern string is specified only variables matching that pattern will be returned.

Syntax Description

info args *(procedure name)*

info body *(procedure name)*

info cmdcount

info commands *(pattern)*

info complete *(commands)*

info default *(procedure name) (argument) (variable name)*

info exists *(variable name)*

info globals *(pattern)*

info level *(number)*

info library

info locals *(pattern)*

info procs *(pattern)*

info script *(filename)*

info tclversion

info vars *(pattern)*

Usage Examples

The following example creates a user defined procedure then returns the contents of that procedure to the screen using the info command.

```
set variable1 5
proc sum {} {
    set variable2 10
    global variable1
    set answer [expr {$variable1 + $variable2}]
    return $answer
}
echo "[sum]"
echo "[info body sum]"
```

Output

```
15

Set variable2 10
global variable1
set answer [expr {$variable1 + $variable2}]
return $answer
```

join

The **join** command takes the provided list *(list name)* and joins each list element together separated by the *(separator)* character. The formatted list is then returned. If no *(separator)* character is provided, then by default the command inserts a space.

Syntax Description

join *(list name) (separator)*

Usage Examples

The following example places the provided separator character between each element of a list and prints the result to the screen.

```
set ip {192 168 1 100}
echo "your IP address is [join ip .]"
```

Output

```
your IP address is 192.168.1.100
```

lappend

The **lappend** command takes the provided values and appends them to the specified *(list name)*. Each of the value arguments is added to the list as its own element separated by a space.

Syntax Description

lappend *(list name) (values)*

Usage Examples

The following example appends a new set of characters to an existing list and prints the list to the screen.

```
set list {1 a 2 b 3 c 4 d}
lappend list 9 8 7 6 5
echo $list
```

Output

```
1 a 2 b 3 c 4 d 9 8 7 6 5
```

linsert

The **linsert** command inserts specified *(elements)* into a specified (list) in the location indicated by the *(index)* value. If no *(index)* is provided or the value is 0, then the *(elements)* are placed at the beginning of the list. If the index is greater than or equal to the number of elements in the list, then the elements will be placed at the end of the list. The **end** option may also be used in place of the index value to indicate that the elements should be appended to the list.

Syntax Description

linsert *(list) (index) (elements)*

linsert *(list)* **end** *(elements)*

Usage Examples

The following example positions two new characters into at the fifth position (That is the fourth indexed starting at 0) and prints that list to the screen.

```
set list {a a a a a a a a}
echo "[linsert $list 4 1 1]"
```

Output

```
a a a a 1 1 a a a a
```

list

The **list** command returns a structured sequence of elements which can be comprised of numbers, strings, or other lists. A list is defined as words or numbers separated by spaces, tabs, or carriage reurns. This command takes care of formatting the output list so it can be correctly parsed by the list specific commands. Information on these list specific commands can be found on the following pages:

lindex on page 17

llength on page 18

lappend on page 58

linsert on page 59

lrange on page 61

lreplace on page 62

lsearch on page 63

lsort on page 64

Syntax Description

list *(elements)*

Usage Examples

The following example creates a list from a string of provided characters.

list 1 a 2 b 3 c 4 d

lrange

The **lrange** command returns a range of elements from a provided Tcl list. This new list is created by referencing the *(first)* and *(last)* indexed places indicating the beginning and ending elements of the new list. If **end** is used in place of the *(last)* argument, then the range will extend through the end of the list. If the *(first)* index argument is invalid, the command will return an empty string.

Syntax Description

lrange *(list) (first) (last)*

lrange *(list) (first)* **end**

Usage Examples

The following example creates a new list by extracting characters from the provided list based on the given range. The new list is then printed to the screen.

```
echo "[lrange {1 a 2 b 3 c 4 d} 3 end]"
```

Output

```
b 3 c 4 d
```

lreplace

The **lreplace** command returns a new list by replacing characters in the original list as indicated by the given indexes. The *(first index)* and *(last index)* positions indicate the beginning and ending elements to be replaced with new elements. A **0** is used to indicate the first element in the list, and the **end** option can be used to replace the last element in the list. If the *(last index)* is less than zero but greater than the *(first index)* then the new elements are placed at the beginning of the list. If no new elements are indicated, then the elements in the specified index positions are deleted.

Syntax Description

```
lreplace (list) (first index) (last index) (new elements)
```

Usage Examples

The following example replaces the two characters in index positions 3 and 4 in the provided list with the new characters (z z). The resulting list is printed to the screen.

```
echo “[lreplace {a 1 b 2 c 3 d 4} 3 4 z z]”
```

Output

```
a 1 b z z 3 d 4
```

lsearch

The **lsearch** command searches the elements of the provided *(list*) for the specified *(pattern)* and returns the indexed position of the first element matching the pattern. If there is no matching element found in the list, a -1 is returned.

Syntax Description

l**search** *(list) (pattern)*

Usage Examples

The following example searches the provided list, returning the index value of the element that matches the search pattern **c**. This result is then printed to the screen.

echo “[lsearch {a 1 b 2 c 3 d 4} c]”

Output

4

lsort

The **lsort** command takes the elements of a specified (list) and returns the elements sorted alphanumerically. First number elements are sorted, from lowest to highest, followed by all Capital letter elements from A to Z, then lowercase letter elements from a to z.

Syntax Description

lsort *(list)*

Usage Examples

The following example sorts the characters from the original list and prints the sorted list to the screen.

```
echo "[lsort {q 0 a bob 3 r 1 n 5 mary l P v}]"
```

Output

```
0 1 3 5 P a bob l mary n q r v
```

proc

The **proc** command creates a Tcl procedure that can be called within the script just like other Tcl commands. A procedure is created with the *(name)* specified. The *(arguments)* necessary for the command are specified next, followed by the *(body)*. When the command is called by *(name)* the *(body)* is executed using any *(arguments)* provided.

Syntax Description

proc *(name) (arguments) (body)*

Usage Examples

The following example creates a procedure for adding two numbers together. The procedure is then run and its results printed to the screen.

```
proc sum {variable1 variable2} {
    set answer [expr {$variable1 + $variable2}]
    return $answer
}
echo "[sum 5 10]"
```

Output

```
15
```

regexp

The **regexp** command compares a given *(regular expression)* against a *(string)*. If the two match then a 1 is returned. If they don't match a 0 is returned. The **–nocase** switch may be used to prevent the interpreter from case sensitive matching. Multiple (match variables) can be appended to the command to store information concerning which parts of the string matched the expression. The first match variable stores the characters of the string that match the leftmost sub-expression. Each subsequent match variable holds the next piece of matching data left to right.

It is necessary to understand regular expressions to make full use of this command. A regular expression is a method for describing the pattern for which a string is being searched. There are a variety of characters provided for pattern matching. See the table below for more information on these characters.

Symbol	**Description**
*****	Matches the largest series (0 or more) of the preceding expression.
+	Matches the largest series (1 or more) of the preceding expression.
?	Indicates (using the Boolean quantifier) that the pattern may or may not occur.
{m}	A sequence with exactly *m* number of matches.
{m,}	A sequence with *m* or more matches.
{m,n}	A sequence no less than *m* and no more than *n* matches.
***?**	Non-greedy form of *. If there is more than 1 match, selects the smallest of the matches.
+?	Non-greedy form of +. If there is more than 1 match, selects the smallest of the matches.
??	Non-greedy form of ?. If there is more than 1 match, selects the smallest of the matches.
{m}?	Non-greedy form of {m}. If there is more than 1 match, selects the smallest of the matches.
^	The following expression only matches when at the beginning of a string.
$	The preceding expression only matches when at the end of a string.
(exp)	*Exp* is a series of regular expression characters and treated as a single entity to be matched. Results are returned in match variable if specified.
(?: exp)	*Exp* is a series of regular expression characters and treated as a single entity to be matched. Results are not returned in a match variable.
()	Matches empty string, returns the match to a variable.
(?:)	Matches empty string, does not return the match to a variable.

Syntax Description

regexp *(regular expression) (string)*

regexp *(regular expression) (string) (match variable)*

regexp –nocase *(regular expression) (string) (match variable)*

Usage Examples

The following example compares the given variable to an expression. The result is stored in a second variable and then compared to two options. The appropriate result is returned to the screen.

```
set variable1 3b
set variable2 [regexp {[0-9]+[a-z]} $variable1]
if {$variable2 == 1} then {
    echo "The string matches the expression"
} else {
    echo "The string does not match the expression"
}
```

Output

```
The string matches the expression
```

regsub

The **regsub** command compares a given *(regular expression)* against a *(string)*. If the two match then a 1 is returned. If they don't match a 0 is returned. The **-nocase** switch may be used to prevent the interpreter from case sensitive matching. In addition to matching patterns it replaces those matches with the *(substitution spec)* argument. Like the **regexp** command the results of the command can be stored in multiple match variables. If the results are not stored in match variables the first substituted value is returned as the result of the command. Only the first match will be substituted unless the **-all** switch is specified.

For more information on regular expressions refer to *regexp* on page 66.

Syntax Description

regsub *(regular expression) (string) (substitution spec)*

regsub *(regular expression) (string) (substitution spec) (match variables)*

regsub -nocase *(regular expression) (string) (substitution spec)*

regsub -all *(regular expression) (string) (substitution spec)*

Usage Examples

The following example matches the given variable against the pattern. The matching string is then replaced by the specified substitution string and the result is saved as another variable. The variables values are then printed to the screen.

```
set variable1 3B
regsub -nocase {[0-9]+[a-z]} $variable1 ZZ variable2
echo "The original input was $variable1 and the new output is $variable2"
```

Output

```
The original input was 3B and the new output is ZZ
```

rename

The rename command renames the specified file. In cases where the file to be renamed or the location which it is to be saved is in any directory other than where the script is run, then a complete path must be specified. The extension used in the file name to be accessed or written to must also be included. The returned value of the command is an empty string.

Syntax Description

rename (old filename) (new filename)

Usage Examples

The following example takes an existing file (testfile.tcl) and give it a new name (testfile2.tcl). The contents of that file are then returned to the screen.

```
write testfile.tcl [echo "This file is a test."]

rename testfile.tcl testfile2.tcl

proc returnfile {filename} {
    eval [read $filename]
}

returnfile testfile2.tcl
```

Output

```
This file is a test.
```

return

The **return** command is used within a procedure to force an immediate return from the procedure to the body of the Tcl script. Any *(arguments)* following the **return** command will be used as the value returned by the procedure.

Syntax Description

return *(arguments)*

Usage Examples

The following example uses the return command to exit the procedure with the returned value of the specified variable. This result is then printed to the screen.

```
proc sum {variable1 variable2} {
    set answer [expr {$variable1 + $variable2}]
    return $answer
}
echo "[sum 5 10]"
```

Output

```
15
```

scan

The **scan** command scans a string based on the provided format and is similar to the ANSI C sscanf procedure. There are a number of different options that can be used to format the string. Using the % symbol and the letter of the desired conversion option, a new string is returned. The result of each conversion can be saved in a variable, however if no variable names are specified, the returned value is the formatted string. If no conversions are performed on the string the command returns a -1.

d Specifies that the input must be a decimal integer, the value is stored or returned as a decimal string

o Specifies that the input must be an octal integer, the value is stored or returned as a decimal string

x Specifies that the input must be a hexadecimal integer, the value is stored or returned as a decimal string

u Specifies that the input must be a decimal integer, the value is stored or returned as an unsigned decimal string

i Specifies that the input must be an integer, the value is determined to be decimal, octal, or hexadecimal and the value is stored or returned as a decimal string

c Specifies that the input is a single character, the binary value of the character is stored or returned as a decimal string

s Specifies that the input consists of all characters up to the next whitespace character, the resulting characters are stored or returned

e or **f** or **g** Specifies that the input must be a floating point number consisting of an optional sign, a string of decimal digits, and an optional exponent. It is stored or returned as a floating point string

[chars] Specifies that the input must be any matching characters. The matching string is stored or returned.

[^chars] Specifies that the input must be any non-matching characters. The non-matching string is stored or returned.

Syntax Description

scan *(string) (format) (variable names)*

Usage Examples

The following example scans the given hexidecimal string and formats the contents as decimal numbers based on the formatting options. The results are stored in three variables that are printed to the screen.

```
set string "#08D03F"
echo "[scan $string "#%2x%2x%2x" 1 2 3]"
echo $1
echo $2
echo $3
```

Output

```
3
8
208
63
```

settime

The **settime** command sets an *(variable name)* to the number of seconds since January 1st, 1970 based on the date input. The provided date/time is formatted as year (yyyy), month (mm), day (dd), hour (hh, 24-hour format), minutes (mm), seconds (ss).

Syntax Description

settime *(variable name) (year) (month) (day) (hour) (minutes) (seconds)*

Usage Examples

The following example prints to the screen an elapsed time in seconds based on the provided input time and date.

```
set time 0
settime time 2007 5 1 8 27 30
echo "$time"
```

Output

```
1178008050
```

split

The **split** command returns a Tcl list generated by splitting the given string at the location of each of the split characters. The split characters are removed from the resulting string. This command is used in conjunction with the **join** command.

Syntax Description

split *(string) (split characters)*

Usage Examples

The following example creates a list from the provided data and prints the resulting list to the screen.

```
echo -n “What is your IP Address?”
set ip [input]
set ip [split $ip .]
echo “The split IP is $ip”
```

Output

```
What is your IP address?
The split IP is 10 23 197 246
```

strtoip

The **strtoip** command is used to convert a standard dotted decimal string *(IP address)* into a Tcl list. The given IP address is formatted and saved in the output variable. All decimal dots are removed and replaced with spaces making each section of the address an element in a Tcl list which is saved as the *(variable name)*. The IP address string is checked for correct formatting. If the formatting is incorrect an empty string is set in *(variable name)*.

Syntax Description

strtoip *(variable name) (IP address)*

Usage Examples

The following example creates a list from the provided IP address and prints the resulting list to the screen.

```
set ipaddress "10.23.197.246"
set ipstring 0
strtoip ipstring $ipaddress
echo "The IP Address $ipaddress is equivalent to the list \"$ipstring\" in Tcl"
```

Output

```
The IP Address 10.23.197.246 is equivalent to the list “10 23 197 246” in Tcl
```

subtract64

The subtract64 command subtracts two number arguments and returns the resulting value. The subtract64 command allows 24-dight (64-bit) unsigned numbers, while the **expr** (refer to page 10) command only allows 10-digit (32-bit) signed numbers.

Syntax Description

subtract64 *(argument 1) (argument 2)*

Usage Examples

The following example subtracts two values and prints the result to the screen.

```
echo "[subtract64 40 27]"
```

Output

```
13
```

trace

The trace command is used to monitor the variables activity within a Tcl script. The variable option initiates a trace on the specified (variable name), while the vdelete option terminates the trace. Specific types of variable activity can be specified using the **r** (read), **w** (write), and **u** (unset) options. The **u** option shows variables that have been permanently deleted. The **vinfo** option returns a list of each trace set on the specified *(variable name)*.

Syntax Description

trace variable *(variable name)* **[r | w | u]** *(Tcl command)*

trace vdelete *(variable name)* **[r | w | u]** *(Tcl command)*

trace vinfo *(variable name)*

Usage Examples

The following example outputs the results of the trace command to the screen for a visual representation of the number of times the incremented **i** variable is read during the course of the script.

```
set i 0
set count 0
trace variable i r {echo $count}
for {} {$i < 3} {incr i } {
    echo "this is a test $i"
    incr count
}
```

Output

```
0 i r
0 i r
this is a test 0
1 i r
1 i r
1 i r
this is a test 1
2 i r
2 i r
2 i r
this is a test 2
```

unset

The **unset** command removes one or more previously set variables. Multiple variable names can be specified to unset more than one variable with a single command. The command returns an empty string.

Syntax Description

unset *(variable name)*

Usage Examples

The following example removes a valid Tcl variable from existence.

```
unset variable1
if {[isset variable1]}{
    echo “you should not see this”
}else{
    echo “the variable was unset”
}
```

Output

```
the variable was unset
```

uplevel

The **uplevel** command executes a script at the specified stack level. If no *(level)* is specified the default value is 1, or one level up from the current level. An integer preceded by the **#** symbol specifies an absolute level whereas an integer value for level without a # symbol indicates a level value relative to the current level. The result of the command is the value of the evaluated *(script body)* at the specified *(level)*.

Syntax Description

uplevel *(level) (script body)*

Usage Examples

The following example creates a procedure that effects a variable at the highest level of the call stack. The variable values are printed to the screen and then the procedure is run, setting the global variable to a different value. That value is then printed to the screen.

```
proc test {} {
    uplevel 1 {set myvariable "yes"}
}
set myvariable "no"
echo “before: $myvariable”

test
echo “after: $myvariable”
```

Output

```
before: no
after: yes
```

upvar

The **upvar** command creates a link from a *(global variable)*, one defined in the global namespace or the previous stack frame, to a *(new variable)*, one defined within a procedure. Any references to the *(new variable)* in the procedure are passed from the *(new variable)* to the *(global variable)*. If no *(level)* within the stack is specified the command defaults to the global level.

Syntax Description

upvar *(level) (global variable) (new variable)*

Usage Examples

The following example creates a procedure that takes a global variable and gives it a new value. The variables value is printed to the screen. Then the procedure runs and the variables new value is printed to the screen.

```
proc test {} {
     upvar 1 variable1 localvariable
     set localvariable "yes"
}
set variable1 "no"
echo $variable1

test
echo $variable1
```

Output

```
no
yes
```

Creating Tcl Scripts in AOS

There are two methods for creating Tcl scripts in AOS. Scripts can be created as individual files or saved inline as part of a units configuration file.

Scripts that are individual files can be created using a plain text editor and then uploaded to the units flash memory. For more information on this process refer to the *Tcl Quick Configuration Guide* on page 90.

Inline scripts are created using the **tcl script** command. They are stored in the unit's configuration, and can be referenced by name. To create an inline script use the **tcl script** command from the Global Configuration mode command prompt with the syntax, **tcl script** <*name*> <*delimiter*>, where <*name*> specifies the name used to reference the Tcl script and <*delimiter*> specifies the character used to terminate the input mode for the specified Tcl script. Once the command has been executed Tcl script commands can be entered to create the script. Once the delimiter character is entered and the enter key is pressed the script is saved. Inline scripts are executed using the same set of commands as scripts saved as individual files.

Running Tcl Scripts in AOS

There are three methods available for running Tcl scripts in AOS. Each of these methods is applicable to particular situations. Understanding all three methods makes it possible to fully utilize the capabilities of AOS Tcl scripts.

The first method for running a Tcl script is encountered when AOS runs **flash provisioning** on a unit. Flash provisioning runs automatically on start up when a CompactFlash card is installed in an AOS unit with CompactFlash capability. Once the unit boots and a CLI session begins, it will look for the *flashprov.txt* file located on the CompactFlash card. This file will indicate the name of a *.biz* file, a startup configuration file, and a Tcl script file named *startup-script*. These files are then transferred to the unit and run. The contents of the Tcl script file are defined by the user and remain on the unit to be run at each bootup. This method makes it simple to create a configuration script that automatically runs allowing simple setup of the unit in the field.

The second method used to run Tcl scripts is the AOS **run tcl** command. The command syntax is **run tcl** <*name*>, where <*name*> is a valid Tcl script file or named inline Tcl script. This method is ideal for testing scripts because it is flexible and can be useful in a variety of different situations.

The third method for running Tcl scripts is based on tracks. Tracks facilitate dynamic configuration changes or system health checking using Tcl scripts. Tracks can indicate the occurrence of a system event that can be used to trigger the execution of a script. To set a Tcl script to run based on a track the **tcl run** command is used with the syntax **tcl run** <*name*> **track** <*track name*> **[on-pass | on-fail]**. For more information about tracks refer to the *Network Monitor Track Configuration Command Set* section of the *AOS Command Reference Guide* available online at ADTRAN's Support Forum at https://supportforums.adtran.com.

Example Scripts

Example Network Configuration Script

In order to effectively illustrate the basics of writing a functional script, we will begin with a simple network configuration script that generates a startup configuration file based on user input provided along the way. The example script, shown in full in *Table 4 on page 82*, uses most of the basic Tcl commands necessary to understand when writing future scripts. It illustrates the basic conventions of programming in general, as well as the specific syntactic rules of Tcl. For reference purposes, line numbers have been added in a column to the left of each line of the script. On the pages following the table, each line is broken down into the use of its commands, its function within the context of the script as a whole, and the syntax necessary for the script to function properly.

This script creates a simple startup configuration file based off a standard configuration used by the fictitious ACME corporation. The only knowledge required by the field technician in this situation is whether or not the unit is using DHCP or static addressing for its public address.

Table 4. Network Configuration Script Example

1	#clear the screen
2	echo ""
3	echo ""
4	echo ""
5	echo ""
6	echo ""
7	echo ""
8	echo ""
9	echo ""
10	echo ""
11	echo ""
12	echo ""
13	echo "ACME configuration script"
14	echo ""
15	
16	#see if they are using DHCP or not
17	echo -n "are you using DHCP? \[y/N\]: "
18	set value [input]
19	if { $value == "yes" || $value == "YES" || $value == "Yes" || $value == "Y" || $value == "y"
20	set dhcp 1

```
} else {
    set dhcp 0
    echo -n "what is the ip address of the public interface? "
    set primaryIp [input]
    echo -n "what is the netmask of the public interface? "
    set primaryMask [input]
    echo -n "what is the gateway ip address of the public interface? "
    set primaryGateway [input]
}

#clear the screen again
echo ""
echo ""
echo ""
echo ""
echo ""
echo ""
#show them their config and write the file to flash
echo "Your configuration is:"
if { $dhcp } {
    echo "eth0/1 Interface: DHCP"
    set addressLine "ip address dhcp"
    set routeLine ""
} else {
    echo "eth0/1 IP Address: $primaryIp"
    echo "eth0/1 Netmask: $primaryMask"
    echo "eth0/1 Gateway IP: $primaryGateway"
    set addressLine "ip address $primaryIp $primaryMask"
    set routeLine "ip route 0.0.0.0 0.0.0.0 $primaryGateway"
}

```

52	set file [read startup-config.base]
53	
54	eval "set file \"$file\""
55	delete startup-config
56	write startup-config $file
57	
58	#extra carrage return because the CLI replaces the last line instead of just making a new one
59	echo ""

The first section of the example script (Lines 1 through 15) simply clear the AOS CLI and introduce the script.

#clear the screen

Line 1 uses the # character to denote that any subsequent text is a comment and will be disregarded by the Tcl interpreter. The purpose of a comment when writing a script is to explain the function of the section of code that follows it. Thorough commenting is a good coding practice. It is particularly important in large, complicated scripts to provide reference points for subsequent programmers.

> NOTE *Any Tcl command can be commented out of the code by preceding it with the # character.*

echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”

Lines 2 through 12 use the Tcl **echo** command followed by the **“ ”** characters. Used with the **“ ”** characters the **echo** command prints an empty string and a carriage return. The script uses Lines 2 through 12 to scroll upward and clear the screen. The **echo** command can be used in conjunction with the **–n** (no new line) option, to suppress the carriage return.

echo “ACME configuration script”

Line 13 uses the Tcl **echo** command to introduce the script by printing the text string **ACME configuration script** to the CLI.

echo “ ”

Line 14 adds another carriage return. The screen has now been cleared and the purpose of our script has been printed to the CLI.

Line 15 is left empty. Leaving a blank line between sections of code is another method of formatting the script for greater legibility.

The second section of the script (Lines 16 through 30) determines, based on user input, whether or not Dynamic Host Configuration Protocol (DHCP) is being used. Depending on user input, the script will continue to prompt the user for more information.

#see if they are using DHCP or not

Line 16 uses the # symbol to create a comment explaining the section of code that follows.

echo –n “are you using DHCP? \[Y/N\]: ”

Line 17 uses the Tcl **echo** command in conjunction with the **–n** option, preceding the text string. The **-n** option suppresses the carriage return and causes the user-input cursor to appear directly after the text string rather than on a new line. Using the backslash character (\) immediately before the bracket characters (**[]**) prevents them from being processed as a special character. They are output instead as part of the text string.

> NOTE *Using the backslash character (\) immediately before any special character prevents it from being processed as a special character. They are output instead as part of the text string. See Table 2 on page 4 for a list of special characters and their functions.*

set value [input]

Line 18 uses the Tcl **set** command to establish the value of the variable named **value**. The Tcl **input** command enclosed in brackets indicates the script is now waiting for user input with the cursor appearing directly after the echoed text from Line 17. The user input text will be stored as a string within the **value** variable.

if { $value = = “yes” || $value = = “YES” || $value = = “Yes” || $value = = “Y” || $value = = “y” } { set dhcp 1

Lines 19 and 20 use the Tcl **if** command to evaluate the user input from Lines 17 and 18 and determine what information needs to be obtained next. The **if** command is issued with a braced expression that checks the user-input text from Line 17 stored in the **value** variable. In order to offer some flexibility and allow users the option of inputting yes in variety of ways, the **logical or** (||) operator is used. If any of the listed **logical or** options are matched in the **value** variable, then the first body argument (**set dhcp 1**) is executed. The argument uses the Tcl **set** command to set the value of the variable **dhcp** to **1**. If none of the **logical or** options are matched then the first body argument is skipped, and the **else** portion of the **if** command is executed.

```
} else {
    set dhcp 0
    echo –n “what is the ip address of the public interface ”
    set primaryIp [input]
    echo –n “what is the netmask of the public interface? ”
    set primaryMask [input]
    echo –n “what is the gateway ip address of the public interface? ”
    set primaryGateway [input]
}
```

Lines 21 through 29 contain the **else** portion of the **if** command in conjunction with the **echo –n** to print several questions to the screen prompting the user for input. The Tcl **set** command in conjunction with the Tcl **input** command is used to set a value for each variable (**primaryIp**, **primaryMask**, and **primaryGateway**) in turn.

It is important to note that when an opening bracket (**[**) or brace (**{**) is used within a command there must be a closing bracket (**]**) or brace (**}**) to match it. The **if** command in the example script use several such brackets and braces. Using line spacing and brace/bracket positioning that mimics the example script and the examples contained in the *Command Reference Guide* will ensure that your scripts are syntactically correct.

NOTE *When an opening bracket or brace is used within a command there must be a closing bracket or brace to match it. For example, the opening bracket of the **else** portion of the **if** command on Line 21 is closed on Line 29.*

Line 30 is left empty for greater legibility.

The third section (Lines 31 through 51) clears the screen again and displays the configuration that has just been stored in the user-input variables in the script. Notice that the author of the script has continued to use thorough commenting throughout.

```
#clear the screen again
```

Line 31 comments that the screen will be cleared again.

```
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
echo “ ”
```

Lines 32 through 37 use the Tcl **echo** command to scroll and clear the screen.

```
#show them their config and write the file to flash
```

Line 38 comments on next few lines of code.

```
echo “Your configuration is:”
```

Line 39 displays the text string to the CLI introducing the user’s configuration.

```
if {$dhcp} {
    echo “eth0/1 Interface: DHCP”
    set addressLine “ip address dhcp”
    set routeLine “ ”
```

Line 40 uses the Tcl **if** command to determine if a **dhcp** variable has been set (refer to Lines 17 through 20). If the variable is set, then the text strings and matching variables from Lines 41 through 43 are displayed. If no **dhcp** variable has been set, then the first body argument is skipped and the **else** portion of the **if** command is executed.

```
} else {
    echo “eth0/1 IP Address: $primaryIp”
    echo “eth0/1 Netmask: $primaryMask”
    echo “eth0/1 Gateway IP: $primaryGateway”
    set addressLine “ip address $primaryIp $primaryMask”
    set routeLine “ip route 0.0.0.0 0.0.0.0 $primaryGateway”
}
```

Lines 44 through 50 display the user’s configuration based on user input received in Lines 23 through 28.

Line 51 is left blank.

This brings us to the final section (Lines 52 through 59) of the script. All the necessary information has been obtained and stored in variables, and the resulting configuration has been displayed to the screen. Now the new configuration must be saved.

```
set file [read startup-config.base]
```

Line 52 uses the Tcl **set** command to set a new variable named **file**. The Tcl **read** command (placed within brackets to allow for command substitution) reads the contents of the external file **startup-config.base** and stores it as a string within the **file** variable. The format of the **startup-config.base** file (shown below) is a standard configuration file with the exception that it contains two variables (recognized by the preceding **$** symbol).

```
int eth 0/1
$addressLine
no shut

$routeLine
end
```

Line 53 is left blank.

eval “set file \”$file\ “ ”

Line 54 uses the Tcl **eval** command to evaluate the contents of the **file** variable using the Tcl interpreter. Using the Tcl **set** command, the resulting contents are stored in the **file** variable. This is done so that the two variables that are referenced in the **startup-config.base** file (**addressLine** and **routeLine**) are replaced with the contents previously stored in those variables (see Lines 48 and 49). The new configuration file is now complete and must be saved.

delete startup-config

Line 55 deletes the old **startup-config** file from the unit using the Tcl **delete** command.

write startup-config $file

Line 56 writes the contents of the **file** variable (set in Line 52) to a new **startup-config** file using the Tcl **write** command.

#extra carriage return because the CLI replaces the last line instead of just making a new one echo “ ”

Lines 57 through 59 leave a blank line for readability, comment on why an extra carriage return was added, and adds the carriage return.

Example VRRP Test Script

The following example script aids in testing the Virtual Router Redundancy Protocol (VRRP) feature available on some AOS products. For more information on VRRP functionality in AOS products, refer to the *VRRP for AOS* configuration guide available online at ADTRAN’s Support Forum at https://supportforums.adtran.com. The VRRP feature allows multiple physical routers to act as a single virtual router in order to diminish loss of connectivity in the event of a router or port failure. To simulate this type of failure this script alternately turns on one Ethernet interface and turns off another as it repeats itself.

Table 5. VRRP Test Script Example

1	#VRRP Test Script
2	for {set i 0} {$i < 10000} {incr i 1} {
3	# turn off the interfaces and output the output of running
4	# this command to the terminal
5	echo [cli {
6	interface eth 0/1
7	shutdown
8	exit
9	interface eth 0/2
10	shutdown
11	exit

```
12            }]
13        sleep 15
14        echo [cli {
15            interface eth 0/1
16                no shutdown
17                exit
18            interface eth 0/2
19                no shutdown
20                exit
21            }]
22        sleep 15
23        echo "In the for loop, and i == $i"
24    }
```

Line 1 uses a comment to introduce the script.

Lines 2 through 24 use a single Tcl **for** loop to repeat the body argument a specified number of times. The first argument enclosed in braces in the **for** command sets the value of the variable named **i** to **0**. The number of repetitions is determined by the value set in the second braced argument or **10000**. Tcl compares the value of the variable **i** and as long as the value is less than 10,000 then it executes the next argument. The last argument instructs Tcl to increment the value of the variable **i** variable by **1**. The body argument is then executed.

Lines 3 and 4 use the # symbol to create comments to explain the function of the AOS commands accessed within the body.

Lines 5 through 12 issue a series of AOS commands to be executed by the Tcl script. The AOS commands in this example shut down the specified interfaces. The Tcl **echo** command in Line 5 is used to display the result of the AOS commands to the CLI as they are being executed.

Multiple AOS commands can be run using a single Tcl **cli** command. It is often necessary to run multiple AOS commands under one **cli** command; for example, running subcommands in different AOS command sets. Each new execution of the **cli** command results in a new occurrence of the CLI. AOS commands that are nested, requiring certain AOS commands to be run before others can be run, must all be run in one **cli** command in order for those dependencies to work.

NOTE *Multiple AOS commands can be run using a single* ***cli*** *command. It is often necessary to execute multiple AOS commands under one Tcl* ***cli*** *command; for example, running subcommands in different AOS command sets. Each new execution of the* ***cli*** *command results in a new occurrence of the CLI.*

Line 13 uses the Tcl **sleep** command to instruct the script to pause for a certain number of seconds. In this case 15 seconds.

Lines 14 through 21 issue AOS commands that reactivate the interfaces deactivated in Lines 5 through 12 and display the result to the CLI.

Line 22 uses the Tcl **sleep** command to pause the script for 15 seconds.

Line 23 displays the text string and the current value of the **i** variable. Each time through the script's **for** loop, the **i** value is increased by 1 until the value equals 10,000. At that point the test fails and the script ends.

Tcl Quick Configuration Guide

1. Open a plain text editor, or execute the **tcl script** command from the Enable mode command prompt. Most operating systems provide a plain text editor. Do not use word processor software for generating Tcl script files. Word processors contain proprietary control characters that can cause the script to function improperly. Alternately, the **copy console** command can be used in the CLI to enter text. For more information on the **copy console** command refer to the *AOS Command Reference Guide* available online at ADTRAN's Support Forum at https://supportforums.adtran.com.
2. Construct a script using the applicable Tcl commands, operators, and special characters. The majority of the work involved in writing the script will occur during this step. Specific instructions for writing the script will vary drastically depending on the purpose of the script. Following the syntactic rules and logical constructs provided in this document will help ensure a properly functioning script.

> NOTE *If you are using the **tcl script** command to generate your script, skip to Step 5.*

3. Save the script. No particular file extension is required for a Tcl file, however it is recommended that a **.tcl** extension be used for the sake of consistency. If the file is intended to be executed by **flash provisioning**, then the file must be named **startup-script**.
4. Upload the script file to the unit. Using a web browser, FTP to the unit using its IP address, and upload the script onto the device. If your web browser does not have FTP capability, a commercial FTP client can be used.

   If the **copy console** command was used to construct the script then this step is not neccessary.

   If the script is intended for **flash provisioning** then it should be loaded onto the unit's CompactFlash card where **flash provisioning** configuration files are stored rather than on the unit.
5. Troubleshoot the script. Run the script by using the AOS **run tcl** command. If the script halts, then the Tcl interpreter will indicate which command caused the error and why it occurred. If this is not enough

information to successfuly locate the error, then it may be neccessary to use the **Tcl cmdtrace** command. Use the plain text editor to correct the script and repeat Step 4.

6. Run the script. At this point the script can be run from the AOS CLI as desired using the **run tcl** command. If the script is to be triggered based on a track then then **run tcl** command should be used with the **track** option as detailed in the section *Running Tcl Scripts in AOS* on page 81.

Troubleshooting

Troubleshooting a Tcl script can be easily handled by those with experience in software development. While Tcl is a straightforward language with relatively simple and logical syntax, there are several problems that can occur. It is possible to have an invalid logical construct that could result in an endless loop or otherwise improperly functioning script. The most likely cause of an error when writing a Tcl script is improper syntax. These errors typically cause the script to halt but can cause other unexpected results.

Tcl has basic error handling functionality built in. When an error occurs the Tcl interpreter halts the code and returns the problem. The interpreter typically provides enough detail to troubleshoot the issue. It will state which command caused the error and what the error was. In line 3 of *Table 6 on page 91* the **$** necessary to access **variable** is missing from the **if** statement. Due to this error the Tcl interpreter will output the following error message:

```
Syntax error in expression “variable == 0”
```

In a simple script such as this, the error information is specific enough to narrow the problem down to the expression on line 3.

Table 6. Troubleshooting Script Example

1	set variable 2
2	
3	if {variable == 0} {
4	echo “Variable is 0”
5	} elseif {$variable == 1} {
6	echo “Variable is 1”
7	} elseif {$variable == 2} {
8	echo “Variable is 2”
9	} else {
10	echo “Variable is not 0, 1, or 2”
11	}

In a situation where a script is stuck in an endless loop, the script can be exited using **<Ctrl+C>**.

The Catch Command

Another method for handling potential errors is to use the **catch** command. The command (*catch* on page 31) allows the section of code bracketed within the command to be parsed and continue running even if an error occurs. It will return a **1** if there is an error within the section of code and a **0** if there is no error. This value is then saved to a variable if specified. The script continues to run, and the error can then be corrected at the programmer's convenience.