



# HXP

## Hexapod Motion Controller



 **Newport®**

**Tcl Manual**

Intaller Pack Version #30002



---

**NOTE**

[Tcl/Tk](#) has been distributed freely for over 10 years and is now used in thousands of applications by companies and individuals worldwide. You are [free to use it however you wish](#), even in commercial applications.

---

©2018 by Newport Corporation, Irvine, CA. All rights reserved.

Original instructions.

No part of this document may be reproduced or copied without the prior written approval of Newport Corporation. This document is provided for information only, and product specifications are subject to change without notice. Any change will be reflected in future publishings.

# Table of Contents

<b>1.0</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Tcl is a String-Based Command Language .....	1
1.1.1	Tcl Commands .....	1
1.2	Tcl script examples .....	2
1.2.1	Hello, World! .....	2
1.2.2	Variables .....	2
1.2.3	Command Substitution.....	3
1.2.4	Math Expressions.....	3
1.2.5	Backslash Substitution .....	4
1.2.6	Grouping with Braces and Double Quotes.....	5
1.2.6.1	Square Brackets Do Not Group.....	5
1.2.6.2	Grouping Before Substitution.....	5
1.2.6.3	Grouping Math Expressions with Braces .....	6
1.2.6.4	More Substitution Examples .....	6
1.2.7	Procedures.....	6
1.2.8	A Factorial Example .....	7
1.2.9	More About Variables.....	9
1.2.9.1	Funny Variable Names .....	9
1.2.9.2	The <code>unset</code> Command.....	9
1.2.9.3	Using <code>info</code> to Find Out About Variables .....	10
1.2.10	More About Math Expressions .....	11
1.2.11	Comments .....	12
1.2.12	Substitution and Grouping Summary.....	13
1.2.13	Fine Points .....	14
1.3	Reference.....	15
1.3.1	Backslash Sequences .....	15
1.3.2	Arithmetic Operators .....	15
1.3.3	Built-in Math Functions.....	15
<b>2.0</b>	<b>Tcl Command Descriptions .....</b>	<b>17</b>
2.1	Tcl - Summary of Tcl Language Syntax.....	17
2.2	after - Execute a Command After a Time Delay .....	20
2.3	append - Append to Variable.....	21
2.4	array - Manipulate Array Variables.....	22
2.5	binary - Insert and Extract Fields from Binary strings.....	24
2.6	break - Abort Looping Command.....	30
2.7	catch - Evaluate Script and Trap Exceptional Returns .....	31
2.8	cd - Change Working Directory .....	32

2.9	clock - Obtain and Manipulate Time .....	33
2.10	close - Close an Open Channel .....	36
2.11	concat - Join Lists Together .....	37
2.12	continue - Skip to the Next Iteration of a Loop .....	38
2.13	eof - Check for End of File Condition on Channel .....	39
2.14	error - Generate an Error .....	40
2.15	eval - Evaluate a Tcl Script .....	41
2.16	exec - Invoke Subprocess(es) .....	42
2.17	exit - End the Application .....	44
2.18	expr - Evaluate an Expression .....	45
2.19	fconfigure - Set and Get Options on a Channel .....	50
2.20	fcopy - Copy Data From One Channel to Another .....	53
2.21	file - Manipulate File Names and Attributes .....	55
2.22	fileevent - Execute a Script When a Channel Becomes Readable or Writable .....	59
2.23	flush - Flush Buffered Output for a Channel .....	61
2.24	for - ``For" Loop .....	62
2.25	foreach - Iterate Over All Elements in One or More Lists .....	63
2.26	format - Format a String in the Style of sprintf .....	64
2.27	gets - Read a Line from a Channel .....	67
2.28	glob - Return Names of Files that Match Patterns .....	68
2.29	global - Access Global Variables .....	69
2.30	if - Execute Scripts Conditionally .....	70
2.31	incr - Increment the Value of a Variable .....	71
2.32	info - Return Information About the State of the Tcl Interpreter .....	72
2.33	join - Create a String by Joining Together List Elements .....	75
2.34	lappend - Append List Elements Onto a Variable .....	76
2.35	lindex - Retrieve an Element From a List .....	77
2.36	linsert - Insert Elements Into a List .....	78
2.37	list - Create a List .....	79
2.38	llength - Count the Number of Elements in a List .....	80
2.39	lrange - Return One or More Adjacent Elements From a List .....	81
2.40	lreplace - Replace Elements in a List With New Elements .....	82
2.41	lsearch - See if a List Contains a Particular Element .....	83
2.42	lsort - Sort the Elements of a List .....	84
2.43	namespace - Create and Manipulate Contexts for Commands and Variables .....	85
2.44	open - Open a File-Based or Command Pipeline Channel .....	92
2.45	pid - Retrieve Process id(s) .....	95
2.46	proc - Create a Tcl Procedure .....	96
2.47	puts - Write to a Channel .....	97
2.48	pwd - Return the Current Working Directory .....	98
2.49	read - Read from a Channel .....	99
2.50	rename - Rename or Delete a Command .....	100
2.51	return - Return from a Procedure .....	101

2.52	scan - Parse String Using Conversion Specifiers in the Style of sscanf .....	103
2.53	seek - Change the Access Position for an Open Channel .....	106
2.54	set - Read and Write Variables .....	107
2.55	socket - Open a TCP Network Connection.....	108
2.56	source - Evaluate a File or Resource as a Tcl Script.....	110
2.57	split - Split a STRING into a PROPER Tcl LIST.....	111
2.58	string - Manipulate Strings .....	112
2.59	subst - Perform Backslash, Command, and Variable Substitutions.....	117
2.60	switch - Evaluate One of Several Scripts, Depending on a Given Value .....	119
2.61	tell - Return Current Access Position for an Open Channel .....	121
2.62	time - Time the Execution of a Script.....	122
2.63	trace - Monitor Variable Accesses, Command Usages and Command Executions .....	123
2.64	unset - Delete Variables.....	128
2.65	update - Process Pending Events and Idle Callbacks.....	129
2.66	uplevel - Execute a Script in a Different Stack Frame.....	130
2.67	upvar - Create Link to Variable in a Different Stack Frame.....	131
2.68	variable - Create and Initialize a Namespace Variable .....	133
2.69	vwait - Process Events Until a Variable is Written.....	134
2.70	while - Execute Script Repeatedly as Long as a Condition is Met .....	135
<b>3.0</b>	<b>TCL commands Not Supported .....</b>	<b>136</b>
<b>4.0</b>	<b>Boot Tcl Script.....</b>	<b>137</b>
<b>5.0</b>	<b>Telnet Connection.....</b>	<b>138</b>
<b>6.0</b>	<b>Error Handling.....</b>	<b>140</b>
<b>7.0</b>	<b>Examples of Tcl Programs with XPS .....</b>	<b>144</b>
7.1	Using analog I/O for Motion .....	144
7.2	Using Digital I/O for Motion.....	147
7.3	Test GPIO1 .....	150
7.4	Gathering with Motion .....	152
7.5	External Gathering.....	155
7.6	Position Compare .....	158
7.7	Master-Slave Mode.....	160
7.8	Jogging .....	162
7.9	Jogging and Gathering.....	164
7.10	Analog Position Tracking .....	168
7.11	Backlash Compensation .....	170
7.12	Timer Event and Global Variables .....	173
7.13	TCL script with input arguments .....	177
	<b>Service Form .....</b>	<b>179</b>





# Hexapod Motion Controller HXP

## 1.0 Introduction

### 1.1 Tcl is a String-Based Command Language

The language has only a few fundamental constructs and relatively little syntax, which makes it easy to learn. The Tcl syntax is meant to be simple. Tcl is designed to be a glue that assembles software building blocks into applications. A simpler glue makes the job easier. In addition, Tcl is interpreted when the application runs. The interpreter makes it easy to build and refine your application

in an interactive manner. A great way to learn Tcl is to try out commands interactively. If you are not sure how to run Tcl on your system, see Chapter 2 for instructions for starting Tcl on UNIX systems. This chapter takes you through the basics of the Tcl language syntax. Even if you are an expert programmer, it is worth taking the time to read these few pages to make sure you understand the fundamentals of Tcl. The basic mechanisms are all related to strings and string substitutions, so it is fairly easy to visualize what is going on in the interpreter. The model is a little different from some other programming languages with which you may already be familiar, so it is worth making sure you understand the basic concepts.

#### 1.1.1 Tcl Commands

Tcl stands for Tool Command Language. A command does something for you, like output a string, compute a math expression, or display a widget on the screen. Tcl casts everything into the mold of a command, even programming constructs like variable assignment and procedure definition. Tcl adds a tiny amount of syntax needed to properly invoke commands, and then it leaves all the hard work up to the command implementation.

The basic syntax for a Tcl command is:

```
command arg1 arg2 arg3 ...
```

The `command` is either the name of a built-in command or a Tcl procedure. White space (i.e., spaces or tabs) is used to separate the command name and its arguments, and a newline (i.e., the end of line character) or semicolon is used to terminate a command. Tcl does not interpret the arguments to the commands except to perform *grouping*, which allows multiple words in one argument, and

*substitution*, which is used with programming variables and nested command calls. The behavior of the Tcl command processor can be summarized in three basic steps:

- Argument grouping.
- Value substitution of nested commands, variables, and backslash escapes.
- Command invocation. It is up to the command to interpret its arguments.

## 1.2 Tcl script examples

### 1.2.1 Hello, World!

**Example 1–1:** The “Hello, World!” example.

```
puts stdout {Hello, World!}
=> Hello, World!
```

In this example, the command is `puts`, which takes two arguments: an I/O stream identifier and a string. `puts` writes the string to the I/O stream along with a trailing newline character.

There are two points to emphasize:

- Arguments are interpreted by the command. In the example, `stdout` is used to identify the standard output stream. The use of `stdout` as a name is a convention employed by `puts` and the other I/O commands. Also, `stderr` is used to identify the standard error output, and `stdin` is used to identify the standard input. Chapter 9 describes how to open other files for I/O.
- Curly braces are used to group words together into a single argument. The `puts` command receives `Hello, World!` as its second argument.

*The braces are not part of the value.*

The braces are syntax for the interpreter, and they get stripped off before the value is passed to the command. Braces group all characters, including newlines and nested braces, until a matching brace is found. Tcl also uses double quotes for grouping. Grouping arguments will be described in more detail later.

### 1.2.2 Variables

The `set` command is used to assign a value to a variable. It takes two arguments:

The first is the name of the variable, and the second is the value. Variable names can be any length, and case *is* significant. In fact, you can use any character in a variable name.

***It is not necessary to declare Tcl variables before you use them.***

The interpreter will create the variable when it is first assigned a value.

The value of a variable is obtained later with the dollar-sign syntax, illustrated in Example 1–2:

**Example 1–2:** Tcl variables.

```
set var 5
=> 5
set b $var
=> 5
```

The second `set` command assigns to variable `b` the value of variable `var`.

The use of the dollar sign is our first example of substitution. You can imagine that the second `set` command gets rewritten by substituting the value of `var` for `$var` to obtain a new command.

```
set b 5
```

The actual implementation of substitution is more efficient, which is important when the value is large.

### 1.2.3 Command Substitution

The second form of substitution is *command substitution*. A nested command is delimited by square brackets, [ ]. The Tcl interpreter takes everything between the brackets and evaluates it as a command. It rewrites the outer command by replacing the square brackets and everything between them with the result of the nested command. This is similar to the use of backquotes in other shells, except that it has the additional advantage of supporting arbitrary nesting of commands.

**Example 1–3:** Command substitution.

```
set len [string length foobar]
=> 6
```

In Example 1–3, the nested command is:

```
string length foobar
```

This command returns the length of the string `foobar`. The nested command runs first. Then, command substitution causes the outer command to be rewritten as if it were:

```
set len 6
```

If there are several cases of command substitution within a single command, the interpreter processes them from left to right. As each right bracket is encountered, the command it delimits is evaluated. This results in a sensible ordering in which nested commands are evaluated first so that their result can be used in arguments to the outer command.

### 1.2.4 Math Expressions

The Tcl interpreter itself does not evaluate math expressions. Tcl just does grouping, substitutions and command invocations. The `expr` command is used to parse and evaluate math expressions.

**Example 1–4:** Simple arithmetic.

```
expr 7.2 / 4
=> 1.8
```

The math syntax supported by `expr` is the same as the C expression syntax. The `expr` command deals with integer, floating point, and boolean values. Logical operations return either 0 (false) or 1 (true). Integer values are promoted to floating point values as needed. Octal values are indicated by a leading zero (e.g., `033` is 27 decimal). Hexadecimal values are indicated by a leading `0x`. Scientific notation for floating point numbers is supported. A summary of the operator precedence is given on page 20.

You can include variable references and nested commands in math expressions.

The following example uses `expr` to add the value of `x` to the length of the string `foobar`. As a result of the innermost command substitution, the `expr` command sees `6 + 7`, and `len` gets the value 13:

**Example 1–5:** Nested commands.

```
set x 7
set len [expr [string length foobar] + $x]
=> 13
```

The expression evaluator supports a number of built-in math functions. Example 1–6 computes the value of `pi`:

**Example 1–6:** Built-in math functions.

```
set pi [expr 2*asin(1.0)]
=> 3.1415926535897931
```

The implementation of `expr` is careful to preserve accurate numeric values and avoid conversions between numbers and strings. However, you can make `expr` operate more efficiently by grouping the entire expression in curly braces. The explanation has to do with the byte code compiler that Tcl uses internally, and its effects are explained in

more detail on page 15. For now, you should be aware that these expressions are all valid and run a bit faster than the examples shown above:

**Example 1–7:** Grouping expressions with braces.

```
expr {7.2 / 4}
set len [expr {[string length foobar] + $x}]
set pi [expr {2*asin(1.0)}]
```

## 1.2.5 Backslash Substitution

The final type of substitution done by the Tcl interpreter is *backslash substitution*. This is used to quote characters that have special meaning to the interpreter. For example, you can specify a literal dollar sign, brace, or bracket by quoting it with a backslash. As a rule, however, if you find yourself using lots of backslashes, there is probably a simpler way to achieve the effect you are striving for. In particular, the `list` command will do quoting for you automatically. In Example 1–8 backslash is used to get a literal `$`:

**Example 1–8:** Quoting special characters with backslash.

```
set dollar \$foo
=> $foo
set x $dollar
=> $foo
```

Only a single round of interpretation is done.

The second `set` command in the example illustrates an important property of Tcl. The value of `dollar` does not affect the substitution performed in the assignment to `x`. In other words, the Tcl parser does not care about the value of a variable when it does the substitution. In the example, the value of `x` and `dollar` is the string `$foo`. In general, you do not have to worry about the value of variables until you use `eval`.

You can also use backslash sequences to specify characters with their Unicode, hexadecimal, or octal value:

```
set escape \u001b
set escape \0x1b
set escape \033
```

The value of variable `escape` is the ASCII ESC character, which has character code 27. The table on page 20 summarizes backslash substitutions.

A common use of backslashes is to continue long commands on multiple lines. This is necessary because a newline terminates a command. The backslash in the next example is required; otherwise the `expr` command gets terminated by the newline after the plus sign.

**Example 1–9:** Continuing long lines with backslashes.

```
set totalLength [expr [string length $one] + \
[string length $two]]
```

There are two fine points to escaping newlines. First, if you are grouping an argument as described in the next section, then you do not need to escape newlines; the newlines are automatically part of the group and do not terminate the command. Second, a backslash as the last character in a line is converted into a space, and all the white space at the beginning of the next line is replaced by this

substitution. In other words, the backslash-newline sequence also consumes all the leading white space on the next line.

## 1.2.6 Grouping with Braces and Double Quotes

Double quotes and curly braces are used to group words together into one argument. The difference between double quotes and curly braces is that quotes allow substitutions to occur in the group, while curly braces prevent substitutions. This rule applies to command, variable, and backslash substitutions.

**Example 1–10:** Grouping with double quotes vs. braces.

```
set s Hello
=> Hello

puts stdout "The length of $s is [string length $s]."
=> The length of Hello is 5.

puts stdout {The length of $s is [string length $s].}
=> The length of $s is [string length $s].
```

In the second command of Example 1–10, the Tcl interpreter does variable and command substitution on the second argument to `puts`. In the third command, substitutions are prevented, so the string is printed as is. In practice, grouping with curly braces is used when substitutions on the argument must be delayed until a later time (or never done at all). Examples include loops, conditional statements, and procedure declarations. Double quotes are useful in simple cases like the `puts` command previously shown.

Another common use of quotes is with the `format` command. This is similar to the C `printf` function. The first argument to `format` is a format specifier that often includes special characters like newlines, tabs, and spaces. The easiest way to specify these characters is with backslash sequences (e.g., `\n` for newline and `\t` for tab). The backslashes must be substituted before the `format` command is called, so you need to use quotes to group the format specifier.

```
puts [format "Item: %s\t%5.3f" $name $value]
```

Here `format` is used to align a name and a value with a tab. The `%s` and `%5.3f` indicate how the remaining arguments to `format` are to be formatted. Note that the trailing `\n` usually found in a C `printf` call is not needed because `puts` provides one for us. For more information about the `format` command.

### 1.2.6.1 Square Brackets Do Not Group

The square bracket syntax used for command substitution does not provide grouping. Instead, a nested command is considered part of the current group. In the command below, the double quotes group the last argument, and the nested command is just part of that group. `puts stdout "The length of $s is [string length $s]."`

If an argument is made up of only a nested command, you do not need to group it with double-quotes because the Tcl parser treats the whole nested command as part of the group.

```
puts stdout [string length $s]
```

The following is a redundant use of double quotes:

```
puts stdout "[expr $x + $y]"
```

### 1.2.6.2 Grouping Before Substitution

The Tcl parser makes a single pass through a command as it makes grouping decisions and performs string substitutions. Grouping decisions are made before substitutions are performed, which is an important property of Tcl. This means that the values being substituted do not affect grouping because the grouping decisions have already been made.

The following example demonstrates how nested command substitution affects grouping. A nested command is treated as an unbroken sequence of characters, regardless of its internal structure. It is included with the surrounding group of characters when collecting arguments for the main command.

**Example 1–11:** Embedded command and variable substitution.

```
set x 7; set y 9
puts stdout $x+$y=[expr $x + $y]
=> 7+9=16
```

In Example 1–11, the second argument to `puts` is:

```
$x+$y=[expr $x + $y]
```

The white space inside the nested command is ignored for the purposes of grouping the argument. By the time Tcl encounters the left bracket, it has already done some variable substitutions to obtain:

```
7+9=
```

When the left bracket is encountered, the interpreter calls itself recursively to evaluate the nested command. Again, the `$x` and `$y` are substituted before calling `expr`. Finally, the result of `expr` is substituted for everything from the left bracket to the right bracket. The `puts` command gets the following as its second argument:

```
7+9=16
```

### **Grouping before substitution.**

The point of this example is that the grouping decision about `puts`'s second argument is made before the command substitution is done. Even if the result of the nested command contained spaces or other special characters, they would be ignored for the purposes of grouping the arguments to the outer command. Grouping and variable substitution interact the same as grouping and command

substitution. Spaces or special characters in variable values do not affect grouping decisions because these decisions are made before the variable values are substituted.

If you want the output to look nicer in the example, with spaces around the `+` and `=`, then you must use double quotes to explicitly group the argument to `puts`:

```
puts stdout "$x + $y = [expr $x + $y]"
```

The double quotes are used for grouping in this case to allow the variable and command substitution on the argument to `puts`.

## **1.2.6.3 Grouping Math Expressions with Braces**

It turns out that `expr` does its own substitutions inside curly braces. This is explained in more detail on page 15. This means you can write commands like the one below and the substitutions on the variables in the expression still occur:

```
puts stdout "$x + $y = [expr {$x + $y}]"
```

## **1.2.6.4 More Substitution Examples**

If you have several substitutions with no white space between them, you can avoid grouping with quotes. The following command sets `concat` to the value of variables `a`, `b`, and `c` all concatenated together:

```
set concat $a$b$c
```

Again, if you want to add spaces, you'll need to use quotes:

```
set concat "$a $b $c"
```

In general, you can place a bracketed command or variable reference anywhere. The following computes a command name:

```
[findCommand $x] arg arg
```

## **1.2.7 Procedures**

Tcl uses the `proc` command to define procedures. Once defined, a Tcl procedure is used just like any of the other built-in Tcl commands. The basic syntax to define a procedure is:

```
proc name arglist body
```

The first argument is the name of the procedure being defined. The second argument is a list of parameters to the procedure. The third argument is a *command body* that is one or more Tcl commands. The procedure name is case sensitive, and in fact it can contain any characters. Procedure names and variable names do not conflict with each

other. As a convention, this book begins procedure names with uppercase letters and it begins variable names with lowercase letters. Good programming style is important as your Tcl scripts get larger.

**Example 1–12:** Defining a procedure.

```
proc Diag {a b} {
  set c [expr sqrt($a * $a + $b * $b)]
  return $c
}
puts "The diagonal of a 3, 4 right triangle is [Diag 3 4]"
```

=> The diagonal of a 3, 4 right triangle is 5.0

The `Diag` procedure defined in the example computes the length of the diagonal side of a right triangle given the lengths of the other two sides. The `sqrt` function is one of many math functions supported by the `expr` command. The variable `c` is local to the procedure; it is defined only during execution of `Diag`. Variable scope is discussed further in Chapter 7. It is not really necessary to use the variable `c` in this example. The procedure can also be written as:

```
proc Diag {a b} {
  return [expr sqrt($a * $a + $b * $b)]
}
```

The `return` command is used to return the result of the procedure. The `return` command is optional in this example because the Tcl interpreter returns the value of the last command in the body as the value of the procedure. So, the procedure could be reduced to:

```
proc Diag {a b} {
  expr sqrt($a * $a + $b * $b)
}
```

Note the stylized use of curly braces in the example. The curly brace at the end of the first line starts the third argument to `proc`, which is the command body. In this case, the Tcl interpreter sees the opening left brace, causing it to ignore newline characters and scan the text until a matching right brace is found. *Double quotes have the same property.* They group characters, including newlines, until another double quote is found. The result of the grouping is that the third argument to `proc` is a sequence of commands. When they are evaluated later, the embedded newlines will terminate each command.

The other crucial effect of the curly braces around the procedure body is to delay any substitutions in the body until the time the procedure is called. For example, the variables `a`, `b`, and `c` are not defined until the procedure is called, so we do not want to do variable substitution at the time `Diag` is defined.

The `proc` command supports additional features such as having variable numbers of arguments and default values for arguments.

## 1.2.8 A Factorial Example

To reinforce what we have learned so far, below is a longer example that uses a `while` loop to compute the factorial function:

**Example 1–13:** A `while` loop to compute factorial.

```
proc Factorial {x} {
  set i 1; set product 1
  while {$i <= $x} {
    set product [expr $product * $i]
    incr i
  }
  return $product
}
Factorial 10
```

=> 3628800

The **semicolon** is used on the first line to remind you that it is a command terminator just like the newline character. The **while** loop is used to multiply all the numbers from one up to the value of `x`. The first argument to **while** is a boolean expression, and its second argument is a command body to execute.

The same math expression evaluator used by the **expr** command is used by **while** to evaluate the boolean expression. There is no need to explicitly use the **expr** command in the first argument to **while**, even if you have a much more complex expression.

The loop body and the procedure body are grouped with curly braces in the same way. The opening curly brace must be on the same line as **proc** and **while**. If you like to put opening curly braces on the line after a **while** or **if** statement, you must escape the newline with a backslash:

```
while {$i < $x} \
{
    set product ...
}
```

*Always group expressions and command bodies with curly braces.*

Curly braces around the boolean expression are crucial because they delay variable substitution until the **while** command implementation tests the expression. The following example is an infinite loop:

```
set i 1; while {$i<=10} {incr i}
```

The loop will run indefinitely.\* The reason is that the Tcl interpreter will substitute for `$i` before **while** is called, so **while** gets a constant expression `1<=10` that will always be true. You can avoid these kinds of errors by adopting a consistent coding style that groups expressions with curly braces:

```
set i 1; while {$i<=10} {incr i}
```

The **incr** command is used to increment the value of the loop variable `i`. This is a handy command that saves us from the longer command:

```
set i [expr $i + 1]
```

The **incr** command can take an additional argument, a positive or negative integer by which to change the value of the variable. Using this form, it is possible to eliminate the loop variable `i` and just modify the parameter `x`. The loop body can be written like this:

```
while {$x > 1} {
    set product [expr $product * $x]
    incr x -1
}
```

Example 1–14 shows factorial again, this time using a recursive definition. A recursive function is one that calls itself to complete its work. Each recursive call decrements `x` by one, and when `x` is one, then the recursion stops.

**Example 1–14:** A recursive definition of factorial.

```
proc Factorial {x} {
    if {$x <= 1} {
        return 1
    } else {
        return [expr $x * [Factorial [expr $x - 1]]]
    }
}
```

## 1.2.9 More About Variables

The `set` command will return the value of a variable if it is only passed a single argument. It treats that argument as a variable name and returns the current value of the variable. The dollar-sign syntax used to get the value of a variable is really just an easy way to use the `set` command. Example 1–15 shows a trick you can play by putting the name of one variable into another variable:

**Example 1–15** Using `set` to return a variable value.

```

set var {the value of var}
=> the value of var
set name var
=> var
set name
=> var
set $name
=> the value of var

```

This is a somewhat tricky example. In the last command, `$name` gets substituted with `var`. Then, the `set` command returns the value of `var`, which is `the value of var`. Nested `set` commands provide another way to achieve a level of indirection. The last `set` command above can be written as follows:

```

set [set name]
=> the value of var

```

Using a variable to store the name of another variable may seem overly complex. However, there are some times when it is very useful. There is even a special command, `upvar`, that makes this sort of trick easier.

### 1.2.9.1 Funny Variable Names

The Tcl interpreter makes some assumptions about variable names that make it easy to embed variable references into other strings. By default, it assumes that variable names contain only letters, digits, and the underscore. The construct `$foo.o` represents a concatenation of the value of `foo` and the literal `".o"`.

If the variable reference is not delimited by punctuation or white space, then you can use curly braces to explicitly delimit the variable name (e.g., `${x}`). You can also use this to reference variables with funny characters in their name, although you probably do not want variables named like that. If you find yourself using funny variable names, or computing the names of variables, then you may want to use the `upvar` command.

**Example 1–16** Embedded variable references.

```

set foo filename
set object $foo.o
=> filename.o
set a AAA
set b abc${a}def
=> abcAAAdef
set .o yuk!
set x ${.o}y
=> yuk!y

```

### 1.2.9.2 The `unset` Command

You can delete a variable with the `unset` command:

```

unset varName varName2 ...

```

Any number of variable names can be passed to the `unset` command. However, `unset` will raise an error if a variable is not already defined.

### 1.2.9.3 Using `info` to Find Out About Variables

The existence of a variable can be tested with the `info exists` command. For example, because `incr` requires that a variable exist, you might have to test for the existence of the variable first.

**Example 1–17:** Using `info` to determine if a variable exists.

```
if {![info exists foobar]} {  
    set foobar 0  
}  
else {  
    incr foobar  
}
```

### 1.2.10 More About Math Expressions

This section describes a few fine points about math in Tcl scripts. In Tcl 7.6 and earlier versions math is not that efficient because of conversions between strings and numbers. The `expr` command must convert its arguments from strings to numbers. It then does all its computations with double precision floating point values. The result is formatted into a string that has, by default, 12 significant digits. This number can be changed by setting the `tcl_precision` variable to the number of significant digits desired. Seventeen digits of precision are enough to ensure that no information is lost when converting back and forth between a string and an IEEE double precision number:

**Example 1–18** Controlling precision with `tcl_precision`.

```
expr 1 / 3
=> 0
expr 1 / 3.0
=> 0.333333333333
set tcl_precision 17
=> 17
expr 1 / 3.0
# The trailing 1 is the IEEE rounding digit
=> 0.3333333333333331
```

In Tcl 8.0 and later versions, the overhead of conversions is eliminated in most cases by the built-in compiler. Even so, Tcl was not designed to support math-intensive applications. There is support for string comparisons by `expr`, so you can test string values in `if` statements. You must use quotes so that `expr` knows to do string comparisons:

```
if {$answer == "yes"} { ... }
```

However, the `string compare` and `string equal` commands more reliable because `expr` may do conversions on strings that look like numbers. Expressions can include variable and command substitutions and still be grouped with curly braces. This is because an argument to `expr` is subject to two rounds of substitution: one by the Tcl interpreter, and a second by `expr` itself. Ordinarily this is not a problem because math values do not contain the characters that are special to the Tcl interpreter. The second round of substitutions is needed to support commands like `while` and `if` that use the expression evaluator internally.

*Grouping expressions can make them run more efficiently.*

You should always group expressions in curly braces and let `expr` do command and variable substitutions. Otherwise, your values may suffer extra conversions from numbers to strings and back to numbers. Not only is this process slow, but the conversions can lose precision in certain circumstances. For example, suppose `x` is computed from a math function:

```
set x [expr {sqrt(2.0)}]
```

At this point the value of `x` is a double-precision floating point value, just as you would expect. If you do this:

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

then you may or may not get 2.0 as the result! This is because Tcl will substitute `$x` and `expr` will concatenate all its arguments into one string, and then parse the expression again. In contrast, if you do this:

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

then `expr` will do the substitutions, and it will be careful to preserve the floating point value of `x`. The expression will be more accurate and run more efficiently because no string conversions will be done. The story behind Tcl values is described in more detail in Chapter 44 on C programming and Tcl.

### 1.2.11 Comments

Tcl uses the pound character, `#`, for comments. Unlike in many other languages, the `#` must occur at the beginning of a command. A `#` that occurs elsewhere is not treated specially. An easy trick to append a comment to the end of a command is to precede the `#` with a **semicolon** to terminate the previous command:

```
# Here are some parameters
set rate 7.0 ;# The interest rate
set months 60 ;# The loan term
```

One subtle effect to watch for is that a **backslash** effectively continues a comment line onto the next line of the script. In addition, a **semicolon** inside a comment is not significant. Only a newline terminates comments:

```
# Here is the start of a Tcl comment \
and some more of it; still in the comment
```

A surprising property of Tcl comments is that curly braces inside comments are still counted for the purposes of finding matching brackets. I think the motivation for this mis-feature was to keep the original Tcl parser simpler. However, it means that the following will not work as expected to comment out an alternate version of an if expression:

```
# if {boolean expression1} {
if {boolean expression2} {
    some commands
}
```

The previous sequence results in an extra left curly brace, and probably a complaint about a missing close brace at the end of your script! A technique I use to comment out large chunks of code is to put the code inside an `if` block that will never execute:

```
if {0} {
    unused code here
}
```

### 1.2.12 Substitution and Grouping Summary

The following rules summarize the fundamental mechanisms of grouping and substitution that are performed by the Tcl interpreter before it invokes a command:

- Command arguments are separated by **white space**, unless arguments are grouped with **curly braces** or **double quotes** as described below.
- Grouping with **curly braces**, **{ }**, prevents substitutions. Braces nest. The interpreter includes all characters between the matching left and right brace in the group, including newlines, semicolons, and nested braces. The enclosing (i.e., outermost) braces are not included in the group's value.
- Grouping with **double quotes**, **" "**, allows substitutions. The interpreter groups everything until another double quote is found, including newlines and semicolons. The enclosing quotes are not included in the group of characters. A double-quote character can be included in the group by quoting it with a backslash, (e.g., `\"`).
- **Grouping decisions** are made before substitutions are performed, which means that the values of variables or command results do not affect grouping.
- A **dollar sign**, **\$**, causes variable substitution. Variable names can be any length, and case is significant. If variable references are embedded into other strings, or if they include characters other than letters, digits, and the underscore, they can be distinguished with the `${varname}` syntax.
- Square **brackets**, **[ ]**, cause command substitution. Everything between the brackets is treated as a command, and everything including the brackets is replaced with the result of the command. Nesting is allowed.
- The **backslash character**, **\**, is used to quote special characters. You can think of this as another form of substitution in which the backslash and the next character or group of characters are replaced with a new character.
- **Substitutions** can occur anywhere unless prevented by curly brace grouping. Part of a group can be a constant string, and other parts of it can be the result of substitutions. Even the command name can be affected by substitutions.
- A single round of substitutions is performed before command invocation. The result of a substitution is not interpreted a second time. This rule is important if you have a variable value or a command result that contains special characters such as spaces, dollar signs, square brackets, or braces. Because only a single round of substitution is done, you do not have to worry about special characters in values causing extra substitutions.

### 1.2.13 Fine Points

- A common error is to forget a space between arguments when grouping with braces or quotes. This is because white space is used as the separator, while the braces or quotes only provide grouping. If you forget the space, you will get syntax errors about unexpected characters after the closing brace or quote. The following is an error because of the missing space between `}` and `{`:

```
if {$x > 1}{puts "x = $x"}
```

- A double quote is only used for grouping when it comes after white space. This means you can include a double quote in the middle of a group without quoting it with a backslash. This requires that curly braces or white space delimit the group. I do not recommend using this obscure feature, but this is what it looks like:

```
set silly a"b
```

- When double quotes are used for grouping, the special effect of curly braces is turned off. Substitutions occur everywhere inside a group formed with double quotes. In the next are

```
set x xvalue
set y "foo {$x} bar"
=> foo {xvalue} bar
```

- When double quotes are used for grouping and a nested command is encountered, the nested command can use double quotes for grouping, too.

```
puts "results [format "%f %f" $x $y]"
```

- Spaces are *not* required around the square brackets used for command substitution. For the purposes of grouping, the interpreter considers everything between the square brackets as part of the current group. The following sets `x` to the concatenation of two command results because there is no space between `]` and `[`.

```
set x [cmd1][cmd2]
```

- Newlines and semicolons are ignored when grouping with braces or double quotes. They get included in the group of characters just like all the others. The following sets `x` to a string that contains newlines:

```
set x "This is line one.
```

```
This is line two.
```

```
This is line three."
```

- During command substitution, newlines and semicolons *are* significant as command terminators. If you have a long command that is nested in square brackets, put a backslash before the newline if you want to continue the command on another line.
- A dollar sign followed by something other than a letter, digit, underscore, or left parenthesis is treated as a literal dollar sign. The following sets `x` to the single character `$`.

```
set x $
```

## 1.3 Reference

### 1.3.1 Backslash Sequences

<code>\a</code>	Bell. (0x7)
<code>\b</code>	Backspace. (0x8)
<code>\f</code>	Form feed. (0xc)
<code>\n</code>	Newline. (0xa)
<code>\r</code>	Carriage return. (0xd)
<code>\t</code>	Tab. (0x9)
<code>\v</code>	Vertical tab. (0xb)
<code>\&lt;newline&gt;</code>	Replace the newline and the leading white space on the next line with a space.
<code>\\</code>	Backslash. ('\')
<code>\ooo</code>	Octal specification of character code. 1, 2, or 3 digits.
<code>\xhh</code>	Hexadecimal specification of character code. 1 or 2 digits.
<code>\uhhhh</code>	Hexadecimal specification of a 16-bit Unicode character value. 4 hex digits.
<code>\c</code>	Replaced with literal <i>c</i> if <i>c</i> is not one of the cases listed above. In particular, <code>\\$, \", \{, \}, \]</code> , and <code>\[</code> are used to obtain these characters.

### 1.3.2 Arithmetic Operators

<code>- ~ !</code>	Unary minus, bitwise NOT, logical NOT.
<code>* / %</code>	Multiply, divide, remainder.
<code>+ -</code>	Add, subtract.
<code>&lt;&lt; &gt;&gt;</code>	Left shift, right shift.
<code>&lt; &gt; &lt;= &gt;=</code>	Comparison: less, greater, less or equal, greater or equal.
<code>== !=</code>	Equal, not equal.
<code>&amp;</code>	Bitwise AND.
<code>^</code>	Bitwise XOR.
<code> </code>	Bitwise OR.
<code>&amp;&amp;</code>	Logical AND.
<code>  </code>	Logical OR.
<code>x?y:z</code>	If <i>x</i> then <i>y</i> else <i>z</i> .

### 1.3.3 Built-in Math Functions

<code>acos (x)</code>	Arccosine of <i>x</i> .
<code>asin (x)</code>	Arsine of <i>x</i> .
<code>atan (x)</code>	Arctangent of <i>x</i> .
<code>atan2 (y, x)</code>	Rectangular ( <i>x, y</i> ) to polar ( <i>r, th</i> ). <code>atan2</code> gives <i>th</i> .
<code>ceil (x)</code>	Least integral value greater than or equal to <i>x</i> .
<code>cos (x)</code>	Cosine of <i>x</i> .
<code>cosh (x)</code>	Hyperbolic cosine of <i>x</i> .
<code>exp (x)</code>	Exponential, <i>e<sup>x</sup></i> .
<code>floor (x)</code>	Greatest integral value less than or equal to <i>x</i> .
<code>fmod (x, y)</code>	Floating point remainder of <i>x/y</i> .
<code>hypot (x, y)</code>	Returns <code>sqrt(x*x + y*y)</code> . <i>r</i> part of polar coordinates.
<code>log (x)</code>	Natural log of <i>x</i> .
<code>log10 (x)</code>	Log base 10 of <i>x</i> .
<code>pow (x, y)</code>	<i>x</i> to the <i>y</i> power, <i>x<sup>y</sup></i> .
<code>sin (x)</code>	Sine of <i>x</i> .
<code>sinh (x)</code>	Hyperbolic sine of <i>x</i> .

<b>sqrt</b> ( <i>x</i> )	Square root of <i>x</i> .
<b>tan</b> ( <i>x</i> )	Tangent of <i>x</i> .
<b>tanh</b> ( <i>x</i> )	Hyperbolic tangent of <i>x</i> .
<b>abs</b> ( <i>x</i> )	Absolute value of <i>x</i> .
<b>double</b> ( <i>x</i> )	Promote <i>x</i> to floating point.
<b>int</b> ( <i>x</i> )	Truncate <i>x</i> to an integer.
<b>round</b> ( <i>x</i> )	Round <i>x</i> to an integer.
<b>rand</b> ()	Return a random floating point value between 0.0 and 1.0.
<b>srand</b> ( <i>x</i> )	Set the seed for the random number generator to the integer <i>x</i> .

## 2.0 Tcl Command Descriptions

### 2.1 Tcl - Summary of Tcl Language Syntax

The following rules define the syntax and semantics of the Tcl language:

[1]

A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

[2]

A command is evaluated in two steps. First, the Tcl interpreter breaks the command into *words* and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

[3]

Words of a command are separated by white space (except for newlines, which are command separators).

[4]

If the first character of a word is double-quote (``"``) then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

[5]

If the first character of a word is an open brace (``{``) then the word is terminated by the matching close brace (``}`). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

[6]

If a word contains an open bracket (``[``) then Tcl performs *command substitution*. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (``]``). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

[7]

If a word contains a dollar-sign (``\$``) then Tcl performs *variable substitution*: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

***\$name***

*Name* is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

***\$name(index)***

*Name* gives the name of an array variable and *index* gives the name of an element within that array. *Name* must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of *index*.

***\${name}***

*Name* is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

[8]

If a backslash (``\``) appears within a word then *backslash substitution* occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

`\a`

Audible alert (bell) (0x7).

`\b`

Backspace (0x8).

`\f`

Form feed (0xc).

`\n`

Newline (0xa).

`\r`

Carriage-return (0xd).

`\t`

Tab (0x9).

`\v`

Vertical tab (0xb).

`\<newline>`*whiteSpace*

A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

`\\`

Backslash (``\``).

`\ooo`

The digits *ooo* (one, two, or three of them) give the octal value of the character.

`\xhh`

The hexadecimal digits *hh* give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

[9]

If a hash character (``#`) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

[10]

Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

[11]

Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

## 2.2 after - Execute a Command After a Time Delay

### Name

**after** - Execute a command after a time delay

### Synopsis

**after** *ms*  
**after** *ms* ?*script script script ...?*  
**after cancel** *id*  
**after cancel** *script script script ...*  
**after idle** ?*script script script ...?*  
**after info** ?*id?*

### Description

This command is used to delay execution of the program or to execute a command in background sometime in the future. It has several forms, depending on the first argument to the command:

**after** *ms*

*Ms* must be an integer giving a time in milliseconds. The command sleeps for *ms* milliseconds and then returns. While the command is sleeping the application does not respond to events.

**after** *ms* ?*script script script ...?*

In this form the command returns immediately, but it arranges for a Tcl command to be executed *ms* milliseconds later as an event handler. The command will be executed exactly once, at the given time. The delayed command is formed by concatenating all the *script* arguments in the same fashion as the **concat** command. The command will be executed at global level (outside the context of any Tcl procedure). The **after** command returns an identifier that can be used to cancel the delayed command using **after cancel**.

**after cancel** *id*

Cancels the execution of a delayed command that was previously scheduled. *Id* indicates which command should be canceled; it must have been the return value from a previous **after** command. If the command given by *id* has already been executed then the **after cancel** command has no effect.

**after cancel** *script script ...*

This command also cancels the execution of a delayed command. The *script* arguments are concatenated together with space separators (just as in the **concat** command). If there is a pending command that matches the string, it is cancelled and will never be executed; if no such command is currently pending then the **after cancel** command has no effect.

**after idle** *script* ?*script script ...?*

Concatenates the *script* arguments together with space separators (just as in the **concat** command), and arranges for the resulting script to be evaluated later as an idle callback. The script will be run exactly once, the next time the event loop is entered and there are no events to process. The command returns an identifier that can be used to cancel the delayed command using **after cancel**.

**after info** ?*id?*

This command returns information about existing event handlers. If no *id* argument is supplied, the command returns a list of the identifiers for all existing event handlers created by the **after** command for this interpreter. If *id* is supplied, it specifies an existing handler; *id* must have been the return value from some previous call to **after** and it must not have triggered yet or been cancelled. In this case the command returns a

list with two elements. The first element of the list is the script associated with *id*, and the second element is either **idle** or **timer** to indicate what kind of event handler it is.

The **after** *ms* and **after idle** forms of the command assume that the application is event driven: the delayed commands will not be executed unless the application enters the event loop. In applications that are not normally event-driven, such as **tclsh**, the event loop can be entered with the **vwait** and **update** commands.

## 2.3 **append - Append to Variable**

### Name

**append** - Append to variable

### Synopsis

**append** *varName* ?*value value value ...*?

### Description

Append all of the *value* arguments to the current value of variable *varName*. If *varName* doesn't exist, it is given a value equal to the concatenation of all the *value* arguments. This command provides an efficient way to build up long variables incrementally.

For example,

**append a \$b**

is much more efficient than

**set a \$a\$b**

if **\$a** is long.

## 2.4 array - Manipulate Array Variables

### Name

**array** - Manipulate array variables

### Synopsis

**array** option arrayName ?arg arg ...?

### Description

This command performs one of several operations on the variable given by *arrayName*. Unless otherwise specified for individual commands below, *arrayName* must be the name of an existing array variable. The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

**array anymore** *arrayName searchId*

Returns 1 if there are any more elements left to be processed in an array search, 0 if all elements have already been returned. *SearchId* indicates which search on *arrayName* to check, and must have been the return value from a previous invocation of **array startsearch**. This option is particularly useful if an array has an element with an empty name, since the return value from **array nextelement** won't indicate whether the search has been completed.

**array donesearch** *arrayName searchId*

This command terminates an array search and destroys all the state associated with that search. *SearchId* indicates which search on *arrayName* to destroy, and must have been the return value from a previous invocation of **array startsearch**. Returns an empty string.

**array exists** *arrayName*

Returns 1 if *arrayName* is an array variable, 0 if there is no variable by that name or if it is a scalar variable.

**array get** *arrayName ?pattern?*

Returns a list containing pairs of elements. The first element in each pair is the name of an element in *arrayName* and the second element of each pair is the value of the array element. The order of the pairs is undefined. If *pattern* is not specified, then all of the elements of the array are included in the result. If *pattern* is specified, then only those elements whose names match *pattern* (using the glob-style matching rules of **string match**) are included. If *arrayName* isn't the name of an array variable, or if the array contains no elements, then an empty list is returned.

**array names** *arrayName ?pattern?*

Returns a list containing the names of all of the elements in the array that match *pattern* (using the glob-style matching rules of **string match**). If *pattern* is omitted then the command returns all of the element names in the array. If there are no (matching) elements in the array, or if *arrayName* isn't the name of an array variable, then an empty string is returned.

**array nextelement** *arrayName searchId*

Returns the name of the next element in *arrayName*, or an empty string if all elements of *arrayName* have already been returned in this search. The *searchId* argument identifies the search, and must have been the return value of an **array startsearch** command. Warning: if elements are added to or deleted from the array, then all searches are automatically terminated just as if **array donesearch** had been invoked; this will cause **array nextelement** operations to fail for those searches.

**array set** *arrayName list*

Sets the values of one or more elements in *arrayName*. *list* must have a form like that returned by **array get**, consisting of an even number of elements. Each odd-numbered element in *list* is treated as an element name within *arrayName*, and the following element in *list* is used as a new value for that array element.

**array size** *arrayName*

Returns a decimal string giving the number of elements in the array. If *arrayName* isn't the name of an array then 0 is returned.

**array startsearch** *arrayName*

This command initializes an element-by-element search through the array given by *arrayName*, such that invocations of the **array nextelement** command will return the names of the individual elements in the array. When the search has been completed, the **array donesearch** command should be invoked. The return value is a search identifier that must be used in **array nextelement** and **array donesearch** commands; it allows multiple searches to be underway simultaneously for the same array.

For example:

```
set personne_info(nom) "André Dupont"  
set personne_info(age) "78"  
set personne_info(metier) "Artiste"  
foreach chose [array names personne_info] {  
    puts "$chose == $personne_info($chose)"  
}  
=> age == 78  
=> metier == Artiste  
=> nom == André Dupont
```

## 2.5 binary - Insert and Extract Fields from Binary strings

### Name

**binary** - Insert and extract fields from binary strings

### Synopsis

**binary format** formatString ?arg arg ...?

**binary scan** string formatString ?varName varName ...?

### Description

This command provides facilities for manipulating binary data.

The first form, **binary format**, creates a binary string from normal Tcl values. For example, given the values 16 and 22, it might produce an 8-byte binary string consisting of two 4-byte integers, one for each of the numbers. The second form of the command, **binary scan**, does the opposite: it extracts data from a binary string and returns it as ordinary Tcl string values.

#### ▪ Binary format

The **binary format** command generates a binary string whose layout is specified by the *formatString* and whose contents come from the additional arguments. The resulting binary value is returned.

The *formatString* consists of a sequence of zero or more field specifiers separated by zero or more spaces. Each field specifier is a single type character followed by an optional numeric *count*. Most field specifiers consume one argument to obtain the value to be formatted. The type character specifies how the value is to be formatted. The *count* typically indicates how many items of the specified type are taken from the value. If present, the *count* is a non-negative decimal integer or \*, which normally indicates that all of the items in the value are to be used. If the number of arguments does not match the number of fields in the format string that consume arguments, then an error is generated.

Each type-count pair moves an imaginary cursor through the binary data, storing bytes at the current position and advancing the cursor to just after the last byte stored. The cursor is initially at position 0 at the beginning of the data. The type may be any one of the following characters:

- a** ..... Stores a character string of length *count* in the output string. If *arg* has fewer than *count* bytes, then additional zero bytes are used to pad out the field. If *arg* is longer than the specified length, the extra characters will be ignored. If *count* is \*, then all of the bytes in *arg* will be formatted. If *count* is omitted, then one character will be formatted. For example,
 

**binary format a7a\*a alpha bravo charlie**

will return a string equivalent to **alpha\000\000bravoc**.
- A** ..... This form is the same as **a** except that spaces are used for padding instead of nulls. For example,
 

**binary format A6A\*A alpha bravo charlie**

will return **alpha bravoc**.
- b** ..... Stores a string of *count* binary digits in low-to-high order within each byte in the output string. *Arg* must contain a sequence of **1** and **0** characters. The resulting bytes are emitted in first to last order with the bits being formatted in low-to-high order within each byte. If *arg* has fewer than *count* digits, then zeros will be used for the remaining bits. If *arg* has more than the specified number of digits, the extra digits will be ignored. If *count* is \*, then all of the digits in *arg* will be

formatted. If *count* is omitted, then one digit will be formatted. If the number of bits formatted does not end at a byte boundary, the remaining bits of the last byte will be zeros. For example,

**binary format b5b\* 11100 111000011010**

will return a string equivalent to `\x07\x87\x05`.

- B** ..... This form is the same as **b** except that the bits are stored in high-to-low order within each byte. For example,

**binary format B5B\* 11100 111000011010**

will return a string equivalent to `\xe0\xe1\xa0`.

- h** ..... Stores a string of *count* hexadecimal digits in low-to-high within each byte in the output string. *Arg* must contain a sequence of characters in the set `"0123456789abcdefABCDEF"`. The resulting bytes are emitted in first to last order with the hex digits being formatted in low-to-high order within each byte. If *arg* has fewer than *count* digits, then zeros will be used for the remaining digits. If *arg* has more than the specified number of digits, the extra digits will be ignored. If *count* is `*`, then all of the digits in *arg* will be formatted. If *count* is omitted, then one digit will be formatted. If the number of digits formatted does not end at a byte boundary, the remaining bits of the last byte will be zeros. For example,

**binary format h3h\* AB def**

will return a string equivalent to `\xba\xed\x0f`.

- H** ..... This form is the same as **h** except that the digits are stored in high-to-low order within each byte. For example,

**binary format H3H\* ab DEF**

will return a string equivalent to `\xab\xde\x0f`.

- c** ..... Stores one or more 8-bit integer values in the output string. If no *count* is specified, then *arg* must consist of an integer value; otherwise *arg* must consist of a list containing at least *count* integer elements. The low-order 8 bits of each integer are stored as a one-byte value at the cursor position. If *count* is `*`, then all of the integers in the list are formatted. If the number of elements in the list is fewer than *count*, then an error is generated. If the number of elements in the list is greater than *count*, then the extra elements are ignored. For example,

**binary format c3cc\* {3 -3 128 1} 257 {2 5}**

will return a string equivalent to `\x03\xfd\x80\x01\x02\x05`, whereas

**binary format c {2 5}**

will generate an error.

- s** ..... This form is the same as **c** except that it stores one or more 16-bit integers in little-endian byte order in the output string. The low-order 16-bits of each integer are stored as a two-byte value at the cursor position with the least significant byte stored first. For example,

**binary format s3 {3 -3 258 1}**

will return a string equivalent to `\x03\x00\xfd\xff\x02\x01`.

- S** ..... This form is the same as **s** except that it stores one or more 16-bit integers in big-endian byte order in the output string. For example,

**binary format S3 {3 -3 258 1}**

will return a string equivalent to `\x00\x03\xff\xfd\x01\x02`.

- i** ..... This form is the same as **c** except that it stores one or more 32-bit integers in little-endian byte order in the output string. The low-order 32-bits of each integer are stored as a four-byte value at the cursor position with the least significant byte stored first. For example,
- binary format i3 {3 -3 65536 1}**
- will return a string equivalent to  
`\x03\x00\x00\x00\xfd\xff\xff\xff\x00\x00\x10\x00`.
- I** ..... This form is the same as **i** except that it stores one or more one or more 32-bit integers in big-endian byte order in the output string. For example,
- binary format I3 {3 -3 65536 1}**
- will return a string equivalent to  
`\x00\x00\x00\x03\xff\xff\xff\xfd\x00\x10\x00\x00`.
- f** ..... This form is the same as **c** except that it stores one or more one or more single-precision floating in the machine's native representation in the output string. This representation is not portable across architectures, so it should not be used to communicate floating point numbers across the network. The size of a floating point number may vary across architectures, so the number of bytes that are generated may vary. If the value is out of range for the machine's native representation, then the value of `FLT_MIN` or `FLT_MAX` as defined by the system will be used instead. Because Tcl uses double-precision floating-point numbers internally, there may be some loss of precision in the conversion to single-precision.
- d** ..... This form is the same as **f** except that it stores one or more one or more double-precision floating in the machine's native representation in the output string.
- x** ..... Stores *count* null bytes in the output string. If *count* is not specified, stores one null byte. If *count* is \*, generates an error. This type does not consume an argument. For example,
- binary format a3xa3x2a3 abc def ghi**
- will return a string equivalent to **abc\000def\000\000ghi**.
- X** ..... Moves the cursor back *count* bytes in the output string. If *count* is \* or is larger than the current cursor position, then the cursor is positioned at location 0 so that the next byte stored will be the first byte in the result string. If *count* is omitted then the cursor is moved back one byte. This type does not consume an argument. For example,
- binary format a3X\*a3X2a3 abc def ghi**
- will return **dghi**.
- @** ..... Moves the cursor to the absolute location in the output string specified by *count*. Position 0 refers to the first byte in the output string. If *count* refers to a position beyond the last byte stored so far, then null bytes will be placed in the uninitialized locations and the cursor will be placed at the specified location. If *count* is \*, then the cursor is moved to the current end of the output string. If *count* is omitted, then an error will be generated. This type does not consume an argument. For example,
- binary format a5@2a1@\*a3@10a1 abcde f ghi j**
- will return **abfdeghi\000\000j**.

- **binary scan**

The **binary scan** command parses fields from a binary string, returning the number of conversions performed. *String* gives the input to be parsed and *formatString* indicates how to parse it. Each *varName* gives the name of a variable; when a field is scanned from *string* the result is assigned to the corresponding variable.

As with **binary format**, the *formatString* consists of a sequence of zero or more field specifiers separated by zero or more spaces. Each field specifier is a single type character followed by an optional numeric *count*. Most field specifiers consume one argument to obtain the variable into which the scanned values should be placed. The type character specifies how the binary data is to be interpreted. The *count* typically indicates how many items of the specified type are taken from the data. If present, the *count* is a non-negative decimal integer or \*, which normally indicates that all of the remaining items in the data are to be used. If there are not enough bytes left after the current cursor position to satisfy the current field specifier, then the corresponding variable is left untouched and **binary scan** returns immediately with the number of variables that were set. If there are not enough arguments for all of the fields in the format string that consume arguments, then an error is generated.

Each type-count pair moves an imaginary cursor through the binary data, reading bytes from the current position. The cursor is initially at position 0 at the beginning of the data. The type may be any one of the following characters:

**a** ..... The data is a character string of length *count*. If *count* is \*, then all of the remaining bytes in *string* will be scanned into the variable. If *count* is omitted, then one character will be scanned. For example,

```
binary scan abcde\000fghi a6a10 var1 var2
```

will return **1** with the string equivalent to **abcde\000** stored in **var1** and **var2** left unmodified.

**A** ..... This form is the same as **a**, except trailing blanks and nulls are stripped from the scanned value before it is stored in the variable. For example,

```
binary scan "abc efghi \000" a* var1
```

will return **1** with **abc efghi** stored in **var1**.

**b** ..... The data is turned into a string of *count* binary digits in low-to-high order represented as a sequence of ``1" and ``0" characters. The data bytes are scanned in first to last order with the bits being taken in low-to-high order within each byte. Any extra bits in the last byte are ignored. If *count* is \*, then all of the remaining bits in **string** will be scanned. If *count* is omitted, then one bit will be scanned. For example,

```
binary scan \x07\x87\x05 b5b* var1 var2
```

will return **2** with **11100** stored in **var1** and **1110000110100000** stored in **var2**.

**B** ..... This form is the same as **B**, except the bits are taken in high-to-low order within each byte. For example,

```
binary scan \x70\x87\x05 b5b* var1 var2
```

will return **2** with **01110** stored in **var1** and **1000011100000101** stored in **var2**.

**h** ..... The data is turned into a string of *count* hexadecimal digits in low-to-high order represented as a sequence of characters in the set ``0123456789abcdef". The data bytes are scanned in first to last order with the hex digits being taken in low-to-high order within each byte. Any extra bits in the last byte are ignored. If *count* is \*, then all of the remaining hex digits in **string** will be scanned. If *count* is omitted, then one hex digit will be scanned. For example,

```
binary scan \x07\x86\x05 h3h* var1 var2
```

will return **2** with **706** stored in **var1** and **50** stored in **var2**.

- H** ..... This form is the same as **h**, except the digits are taken in low-to-high order within each byte. For example,  
**binary scan \x07\x86\x05 H3H\* var1 var2**  
 will return **2** with **078** stored in **var1** and **05** stored in **var2**.
- c** ..... The data is turned into *count* 8-bit signed integers and stored in the corresponding variable as a list. If *count* is *\**, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one 8-bit integer will be scanned. For example,  
**binary scan \x07\x86\x05 c2c\* var1 var2**  
 will return **2** with **7 -122** stored in **var1** and **5** stored in **var2**. Note that the integers returned are signed, but they can be converted to unsigned 8-bit quantities using an expression like:  
**expr ( \$num + 0x100 ) % 0x100**
- s** ..... The data is interpreted as *count* 16-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is *\**, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one 16-bit integer will be scanned. For example,  
**binary scan \x05\x00\x07\x00\xff s2s\* var1 var2**  
 will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**. Note that the integers returned are signed, but they can be converted to unsigned 16-bit quantities using an expression like:  
**expr ( \$num + 0x10000 ) % 0x10000**
- S** ..... This form is the same as **s** except that the data is interpreted as *count* 16-bit signed integers represented in big-endian byte order. For example,  
**binary scan \x00\x05\x00\x07\xff S2S\* var1 var2**  
 will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**.
- i** ..... The data is interpreted as *count* 32-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is *\**, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one 32-bit integer will be scanned. For example,  
**binary scan \x05\x00\x00\x00\x07\x00\x00\x00\xff\xff\xff i2i\* var1 var2**  
 will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**. Note that the integers returned are signed and cannot be represented by Tcl as unsigned values.
- I** ..... This form is the same as **I** except that the data is interpreted as *count* 32-bit signed integers represented in big-endian byte order. For example,  
**binary \x00\x00\x00\x05\x00\x00\x00\x07\xff\xff\xff I2I\* var1 var2**  
 will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**.
- f** ..... The data is interpreted as *count* single-precision floating point numbers in the machine's native representation. The floating point numbers are stored in the corresponding variable as a list. If *count* is *\**, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one single-precision floating point number will be scanned. The size of a floating point number may vary across architectures, so the number of bytes that are scanned may vary. If the data does not represent a valid floating point number, the resulting value is undefined and compiler dependent.
- d** ..... This form is the same as **f** except that the data is interpreted as *count* double-precision floating point numbers in the machine's native representation.
- x** ..... Moves the cursor forward *count* bytes in *string*. If *count* is *\** or is larger than the number of bytes after the current cursor position, then the cursor is positioned after the last byte in *string*. If *count* is omitted, then the cursor is

moved forward one byte. Note that this type does not consume an argument. For example,

```
binary scan \x01\x02\x03\x04 x2H* var1
```

will return **1** with **0304** stored in **var1**.

**X** ..... Moves the cursor back *count* bytes in *string*. If *count* is *\** or is larger than the current cursor position, then the cursor is positioned at location 0 so that the next byte scanned will be the first byte in *string*. If *count* is omitted then the cursor is moved back one byte. Note that this type does not consume an argument. For example,

```
binary scan \x01\x02\x03\x04 c2XH* var1 var2
```

will return **2** with **1 2** stored in **var1** and **020304** stored in **var2**.

**@** ..... Moves the cursor to the absolute location in the data string specified by *count*. Note that position 0 refers to the first byte in *string*. If *count* refers to a position beyond the end of *string*, then the cursor is positioned after the last byte. If *count* is omitted, then an error will be generated. For example,

```
binary scan \x01\x02\x03\x04 c2@1H* var1 var2
```

will return **2** with **1 2** stored in **var1** and **020304** stored in **var2**.

### Platform issues

Sometimes it is desirable to format or scan integer values in the native byte order for the machine. Refer to the **byteOrder** element of the **tcl\_platform** array to decide which type character to use when formatting or scanning integers.

### See also

**format**, **scan**, **tclvars**

## 2.6 **break - Abort Looping Command**

### Name

**break** - Abort looping command

### Synopsis

**break**

### Description

This command is typically invoked inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL\_BREAK code, which causes a break exception to occur. The exception causes the current script to be aborted out to the innermost containing loop command, which then aborts its execution and returns normally. Break exceptions are also handled in a few other situations, such as the **catch** command, Tk event bindings, and the outermost scripts of procedure bodies.

## 2.7 **catch - Evaluate Script and Trap Exceptional Returns**

### Name

**catch** - Evaluate script and trap exceptional returns

### Synopsis

**catch** *script* ?*varName*?

### Description

The **catch** command may be used to prevent errors from aborting command interpretation. **catch** calls the Tcl interpreter recursively to execute *script*, and always returns a TCL\_OK code, regardless of any errors that might occur while executing *script*. The return value from **catch** is a decimal string giving the code returned by the Tcl interpreter after executing *script*. This will be **0** (TCL\_OK) if there were no errors in *script*; otherwise it will have a non-zero value corresponding to one of the exceptional return codes (see tcl.h for the definitions of code values). If the *varName* argument is given, then it gives the name of a variable; **catch** will set the variable to the string returned from *script* (either a result or an error message).

Note that **catch** catches all exceptions, including those generated by **break** and **continue** as well as errors.

## 2.8 **cd - Change Working Directory**

### **Name**

**cd** - Change working directory

### **Synopsis**

**cd** *?dirName?*

### **Description**

Change the current working directory to *dirName*, or to the home directory (as specified in the HOME environment variable) if *dirName* is not given. Returns an empty string.

## 2.9 clock - Obtain and Manipulate Time

### Name

**clock** - Obtain and manipulate time

### Synopsis

**clock** *option* ?*arg arg ...?*

### Description

This command performs one of several operations that may obtain or manipulate strings or values that represent some notion of time. The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

#### **clock clicks**

Return a high-resolution time value as a system-dependent integer value. The unit of the value is system-dependent but should be the highest resolution clock available on the system such as a CPU cycle counter. This value should only be used for the relative measurement of elapsed time.

#### **clock format** *clockValue* ?-**format** *string?* ?-**gmt** *boolean?*

Converts an integer time value, typically returned by **clock seconds**, **clock scan**, or the **atime**, **mtime**, or **ctime** options of the **file** command, to human-readable form. If the **-format** argument is present the next argument is a string that describes how the date and time are to be formatted.

Field descriptors consist of a **%** followed by a field descriptor character. All other characters are copied into the result. Valid field descriptors are:

- %%** Insert a %.
- %a** Abbreviated weekday name (Mon, Tue, etc.).
- %A** Full weekday name (Monday, Tuesday, etc.).
- %b** Abbreviated month name (Jan, Feb, etc.).
- %B** Full month name.
- %c** Locale specific date and time.
- %d** Day of month (01 - 31).
- %H** Hour in 24-hour format (00 - 23).
- %I** Hour in 12-hour format (00 - 12).
- %j** Day of year (001 - 366).
- %m** Month number (01 - 12).
- %M** Minute (00 - 59).
- %p** AM/PM indicator.
- %S** Seconds (00 - 59).
- %U** Week of year (01 - 52), Sunday is the first day of the week.
- %w** Weekday number (Sunday = 0).
- %W** Week of year (01 - 52), Monday is the first day of the week.
- %x** Locale specific date format.
- %X** Locale specific time format.
- %y** Year without century (00 - 99).
- %Y** Year with century (e.g. 1990)

**%Z** Time zone name.

In addition, the following field descriptors may be supported on some systems (e.g.Unix):

**%D** Date as %m/%d/%y.

**%e** Day of month (1 - 31), no leading zeros.

**%h** Abbreviated month name.

**%n** Insert a newline.

**%r** Time as %I:%M:%S %p.

**%R** Time as %H:%M.

**%t** Insert a tab.

**%T** Time as %H:%M:%S.

If the **-format** argument is not specified, the format string "**%a %b %d %H:%M:%S %Z %Y**" is used. If the **-gmt** argument is present the next argument must be a boolean which if true specifies that the time will be formatted as Greenwich Mean Time. If false then the local timezone will be used as defined by the operating environment.

**clock scan** *dateString* **?-base** *clockVal* **?-gmt** *boolean*?

Convert *dateString* to an integer clock value (see **clock seconds**). This command can parse and convert virtually any standard date and/or time string, which can include standard time zone mnemonics. If only a time is specified, the current date is assumed. If the string does not contain a time zone mnemonic, the local time zone is assumed, unless the **-gmt** argument is true, in which case the clock value is calculated assuming that the specified time is relative to Greenwich Mean Time.

If the **-base** flag is specified, the next argument should contain an integer clock value. Only the date in this value is used, not the time. This is useful for determining the time on a specific day or doing other date-relative conversions.

The *dateString* consists of zero or more specifications of the following form:

*time*

A time of day, which is of the form: *hh?:mm?:ss?? ?meridian? ?zone?* or *hhmm ?meridian? ?zone?*. If no meridian is specified, *hh* is interpreted on a 24-hour clock.

*date*

A specific month and day with optional year. The acceptable formats are *mm/dd?/yy?*, *monthname dd ?, yy?*, *dd monthname ?yy?* and *day, dd monthname yy*. The default year is the current year. If the year is less than 100, we treat the years 00-38 as 2000-2038 and the years 70-99 as 1970-1999. The years 39-70 are undefined and may not be valid on certain platforms. (For those platforms where it is defined then the years 69-99 match to 1969-1999.)

*relative time*

A specification relative to the current time. The format is *number unit* acceptable units are **year**, **fortnight**, **month**, **week**, **day**, **hour**, **minute** (or **min**), and **second** (or **sec**). The unit can be specified as a singular or plural, as in **3 weeks**. These modifiers may also be specified: **tomorrow**, **yesterday**, **today**, **now**, **last**, **this**, **next**, **ago**.

The actual date is calculated according to the following steps. First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added. Next, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences and the correct date is given when going from the end of a long month to a short month.

**clock seconds**

Return the current date and time as a system-dependent integer value. The unit of the value is seconds, allowing it to be used for relative time calculations. The value is usually defined as total elapsed time from an ``epoch''. You shouldn't assume the value of the epoch.

## 2.10 **close** - Close an Open Channel

### Name

**close** - Close an open channel.

### Synopsis

**close** *channelId*

### Description

Closes the channel given by *channelId*. *ChannelId* must be a channel identifier such as the return value from a previous **open** or **socket** command. All buffered output is flushed to the channel's output device, any buffered input is discarded, the underlying file or device is closed, and *channelId* becomes unavailable for use.

If the channel is blocking, the command does not return until all output is flushed. If the channel is nonblocking and there is unflushed output, the channel remains open and the command returns immediately; output will be flushed in the background and the channel will be closed when all the flushing is complete.

If *channelId* is a blocking channel for a command pipeline then **close** waits for the child processes to complete.

If the channel is shared between interpreters, then **close** makes *channelId* unavailable in the invoking interpreter but has no other effect until all of the sharing interpreters have closed the channel. When the last interpreter in which the channel is registered invokes **close**, the cleanup actions described above occur. See the **interp** command for a description of channel sharing.

Channels are automatically closed when an interpreter is destroyed and when the process exits. Channels are switched to blocking mode, to ensure that all output is correctly flushed before the process exits.

The command returns an empty string, and may generate an error if an error occurs while flushing output.

## 2.11 concat - Join Lists Together

### Name

**concat** - Join lists together

### Synopsis

**concat** ?arg arg ...?

### Description

This command treats each argument as a list and concatenates them into a single list. It also eliminates leading and trailing spaces in the *arg*'s and adds a single separator space between *arg*'s. It permits any number of arguments.

For example, the command

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

as its result.

If no *args* are supplied, the result is an empty string.

For example:

```
set simple_list "John Joe Mary Susan"  
set simple_liste2 "Michael Samuel Sophie Stéphanie"  
set maliste "Mercure Venus Terre Mars Jupiter"  
set groupe_liste [list $simple_liste $simple_liste2]  
set newliste [ concat $maliste $groupe_liste ]  
puts $newliste
```

```
=> Mercure Venus Terre Mars Jupiter {John Joe Mary Susan} {Michael Samuel  
Sophie Stéphanie}
```

## 2.12 **continue** - Skip to the Next Iteration of a Loop

### Name

**continue** - Skip to the next iteration of a loop

### Synopsis

**continue**

### Description

This command is typically invoked inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL\_CONTINUE code, which causes a continue exception to occur. The exception causes the current script to be aborted out to the innermost containing loop command, which then continues with the next iteration of the loop. Catch exceptions are also handled in a few other situations, such as the **catch** command and the outermost scripts of procedure bodies.

## 2.13 eof - Check for End of File Condition on Channel

### Name

**eof** - Check for end of file condition on channel

### Synopsis

**eof** *channelId*

### Description

Returns 1 if an end of file condition occurred during the most recent input operation on *channelId* (such as **gets**), 0 otherwise.

## 2.14 error - Generate an Error

### Name

**error** - Generate an error

### Synopsis

**error** *message* *?info?* *?code?*

### Description

Returns a `TCL_ERROR` code, which causes command interpretation to be unwound. *Message* is a string that is returned to the application to indicate what went wrong.

If the *info* argument is provided and is non-empty, it is used to initialize the global variable **errorInfo**. **errorInfo** is used to accumulate a stack trace of what was in progress when an error occurred; as nested commands unwind, the Tcl interpreter adds information to **errorInfo**. If the *info* argument is present, it is used to initialize **errorInfo** and the first increment of unwind information will not be added by the Tcl interpreter. In other words, the command containing the **error** command will not appear in **errorInfo**; in its place will be *info*. This feature is most useful in conjunction with the **catch** command: if a caught error cannot be handled successfully, *info* can be used to return a stack trace reflecting the original point of occurrence of the error:

```
catch {...} errMsg
set savedInfo $errorInfo
...
error $errMsg $savedInfo
```

If the *code* argument is present, then its value is stored in the **errorCode** global variable. This variable is intended to hold a machine-readable description of the error in cases where such information is available; see the **tclvars** manual page for information on the proper format for the variable. If the *code* argument is not present, then **errorCode** is automatically reset to ```NONE``` by the Tcl interpreter as part of processing the error generated by the command.

## 2.15 **eval - Evaluate a Tcl Script**

### **Name**

**eval** - Evaluate a Tcl script

### **Synopsis**

**eval** *arg ?arg ...?*

### **Description**

**eval** takes one or more arguments, which together comprise a Tcl script containing one or more commands. **eval** concatenates all its arguments in the same fashion as the **concat** command, passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it).

## 2.16 `exec` - Invoke Subprocess(es)

### Name

`exec` - Invoke subprocess(es)

### Synopsis

`exec` *?switches?* *arg* *?arg ...?*

### Description

This command treats its arguments as the specification of one or more subprocesses to execute. The arguments take the form of a standard shell pipeline where each *arg* becomes one word of a command, and each distinct command becomes a subprocess.

If the initial arguments to `exec` start with `-` then they are treated as command-line switches and are not part of the pipeline specification. The following *switches* are currently supported:

- `-keepnewline` .....Retains a trailing newline in the pipeline's output. Normally a trailing newline will be deleted.
- `--` .....Marks the end of switches. The argument following this one will be treated as the first *arg* even if it starts with a `-`.

If an *arg* (or pair of *arg*'s) has one of the forms described below then it is used by `exec` to control the flow of input and output among the subprocess(es). Such arguments will not be passed to the subprocess(es). In forms such as ``< fileName" fileName` may either be in a separate argument from ``<"` or in the same argument with no intervening space (i.e. ``<fileName"`).

- `|` .....Separates distinct commands in the pipeline. The standard output of the preceding command will be piped into the standard input of the next command.
- `|&` .....Separates distinct commands in the pipeline. Both standard output and standard error of the preceding command will be piped into the standard input of the next command. This form of redirection overrides forms such as `2>` and `>&`.
- `< fileName` .....The file named by *fileName* is opened and used as the standard input for the first command in the pipeline.
- `<@ fileId` .....*FileId* must be the identifier for an open file, such as the return value from a previous call to `open`. It is used as the standard input for the first command in the pipeline. *FileId* must have been opened for reading.
- `<< value` .....*Value* is passed to the first command as its standard input.
- `> fileName` .....Standard output from the last command is redirected to the file named *fileName*, overwriting its previous contents.
- `2> fileName` .....Standard error from all commands in the pipeline is redirected to the file named *fileName*, overwriting its previous contents.
- `>& fileName` .....Both standard output from the last command and standard error from all commands are redirected to the file named *fileName*, overwriting its previous contents.
- `>> fileName` .....Standard output from the last command is redirected to the file named *fileName*, appending to it rather than overwriting it.
- `2>> fileName` .....Standard error from all commands in the pipeline is redirected to the file named *fileName*, appending to it rather than overwriting it.

- >>&*fileName* ..... Both standard output from the last command and standard error from all commands are redirected to the file named *fileName*, appending to it rather than overwriting it.
- >@*fileId* ..... *fileId* must be the identifier for an open file, such as the return value from a previous call to **open**. Standard output from the last command is redirected to *fileId*'s file, which must have been opened for writing.
- 2>@*fileId* ..... *fileId* must be the identifier for an open file, such as the return value from a previous call to **open**. Standard error from all commands in the pipeline is redirected to *fileId*'s file. The file must have been opened for writing.
- >&@*fileId* ..... *fileId* must be the identifier for an open file, such as the return value from a previous call to **open**. Both standard output from the last command and standard error from all commands are redirected to *fileId*'s file. The file must have been opened for writing.

If standard output has not been redirected then the **exec** command returns the standard output from the last command in the pipeline. If any of the commands in the pipeline exit abnormally or are killed or suspended, then **exec** will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the **errorCode** variable will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file and that standard error isn't redirected, then **exec** will return an error; the error message will include the pipeline's standard output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is normally deleted from the result or error message. This is consistent with other Tcl return values, which don't normally end with newlines. However, if **-keepnewline** is specified then the trailing newline is retained.

If standard input isn't redirected with `<` or `<<` or `<@` then the standard input for the first command in the pipeline is taken from the application's current standard input.

If the last *arg* is `&` then the pipeline will be executed in background. In this case the **exec** command will return a list whose elements are the process identifiers for all of the subprocesses in the pipeline. The standard output from the last command in the pipeline will go to the application's standard output if it hasn't been redirected, and error output from all of the commands in the pipeline will go to the application's standard error file unless redirected.

The first word in each command is taken as the command name; tilde-substitution is performed on it, and if the result contains no slashes then the directories in the PATH environment variable are searched for an executable by the given name. If the name contains a slash then it must refer to an executable reachable from the current directory. No `glob` expansion or other shell-like substitutions are performed on the arguments to commands.

### **Portability issues**

The **exec** command is fully functional and works as described.

### **See also**

**open**

## 2.17 **exit - End the Application**

### **Name**

**exit** - End the application

### **Synopsis**

**exit** ?*returnCode*?

### **Description**

Terminate the process, returning *returnCode* to the system as the exit status. If *returnCode* isn't specified then it defaults to 0.

## 2.18 `expr` - Evaluate an Expression

### Name

`expr` - Evaluate an expression

### Synopsis

`expr arg ?arg arg ...?`

### Description

Concatenates *arg*'s (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value. The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

`expr 8.2 + 6`

evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons.

### Operands

A Tcl expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression's instructions. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is **0**), or in hexadecimal (if the first two characters of the operand are **0x**). If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the **f**, **F**, **l**, and **L** suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

[1]

As an numeric value, either integer or floating-point.

[2]

As a Tcl variable, using standard **\$** notation. The variable's value will be used as the operand.

[3]

As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand

[4]

As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.

[5]

As a Tcl command enclosed in brackets. The command will be executed and its result will be used as the operand.

[6]

As a mathematical function whose arguments have any of the above forms for operands, such as **sin(\$x)**. See below for a list of defined functions.

Where substitutions occur above (e.g. inside quoted strings), they are performed by the expression's instructions. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents.

For some examples of simple expressions, suppose the variable **a** has the value 3 and the variable **b** has the value 6. Then the command on the left side of each of the lines below will produce the value on the right side of the line:

```
expr 3.1 + $a      6.1
expr 2 + "$a.$b"  5.6
expr 4*[length "6 2"] 8
expr {{word one} < "word $a"} 0
```

## Operators

The valid operators are listed below, grouped in decreasing order of precedence:

- + ~ !

Unary minus, unary plus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.

\* / %

Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.

+ -

Add and subtract. Valid for any numeric operands.

<<>>

Left and right shift. Valid for integer operands only. A right shift always propagates the sign bit.

< > <= >=

Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

== !=

Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

&

Bit-wise AND. Valid for integer operands only.

^

Bit-wise exclusive OR. Valid for integer operands only.

|

Bit-wise OR. Valid for integer operands only.

&&

Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for numeric operands only (integers or floating-point).

||

Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for numeric operands only (integers or floating-point).

$x?y:z$

If-then-else, as in C. If  $x$  evaluates to non-zero, then the result is the value of  $y$ . Otherwise the result is the value of  $z$ . The  $x$  operand must have a numeric value.

See the C manual for more details on the results produced by each operator. All of the binary operators group left-to-right within the same precedence level.

For example, the command

```
expr 4*2 < 7
```

returns 0.

The **&&**, **||**, and **?:** operators have "lazy evaluation", just as in C, which means that operands are not evaluated if they are not needed to determine the outcome.

For example, in the command

```
expr {$v ? [a] : [b]}
```

only one of **[a]** or **[b]** will actually be evaluated, depending on the value of **\$v**. Note, however, that this is only true if the entire expression is enclosed in braces; otherwise the Tcl parser will evaluate both **[a]** and **[b]** before invoking the **expr** command.

### Math functions

Tcl supports the following mathematical functions in expressions:

<b>acos</b>	<b>cos</b>	<b>hypot</b>	<b>sinh</b>
<b>asin</b>	<b>cosh</b>	<b>log</b>	<b>sqrt</b>
<b>atan</b>	<b>exp</b>	<b>log10</b>	<b>tan</b>
<b>atan2</b>	<b>floor</b>	<b>pow</b>	<b>tanh</b>
<b>ceil</b>	<b>fmod</b>	<b>sin</b>	

Each of these functions invokes the math library function of the same name; see the manual entries for the library functions for details on what they do. Tcl also implements the following functions for conversion between integers and floating-point numbers and the generation of random numbers:

**abs(*arg*)** .....Returns the absolute value of *arg*. *Arg* may be either integer or floating-point, and the result is returned in the same form.

**double(*arg*)** .....If *arg* is a floating value, returns *arg*, otherwise converts *arg* to floating and returns the converted value.

**int(*arg*)** .....If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by truncation and returns the converted value.

**rand()** .....Returns a floating point number from zero to just less than one or, in mathematical terms, the range [0,1). The seed comes from the internal clock of the machine or may be set manual with the **srand** function.

**round(*arg*)** .....If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by rounding and returns the converted value.

**srand(*arg*)** .....The *arg*, which must be an integer, is used to reset the seed for the random number generator. Returns the first random number from that seed. Each interpreter has it's own seed.

In addition to these predefined functions, applications may define additional functions using **Tcl\_CreateMathFunc()**.

### Types, overflow, and precision

All internal computations involving integers are done with the C type *long*, and all internal computations involving floating-point are done with the C type *double*. When converting a string to floating-point, exponent overflow is detected and results in a Tcl error.

For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, integer overflow and underflow are generally not detected reliably for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used.

For example,

```
expr 5 / 4
```

returns **1**, while

```
expr 5 / 4.0
```

```
expr 5 / ([string length "abcd"] + 0.0 )
```

both return **1.25**. Floating-point values are always returned with a ``.`` or an `e` so that they will not look like integer values.

For example,

```
expr 20.0/5.0
```

returns **4.0**, not **4**.

### String operations

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C *sprintf* format specifier `%d` for integers and `%g` for floating-point values.

For example, the commands

```
expr {"0x03" > "2"}
```

```
expr {"0y" < "0x12"}
```

both return **1**. The first comparison is done using integer comparison, and the second is done using string comparison after the second operand is converted to the string **18**. Because of Tcl's tendency to treat values as numbers whenever possible, it isn't generally a good idea to use operators like `==` when you really want string comparison and the values of the operands could be arbitrary; it's better in these cases to use the **string compare** command instead.

### Performance considerations

Enclose expressions in braces for the best speed and the smallest storage requirements. This allows the Tcl bytecode compiler to generate the best code.

As mentioned above, expressions are substituted twice: once by the Tcl parser and once by the **expr** command.

For example, the commands

```
set a 3
set b {$a + 2}
expr $b*4
```

return 11, not a multiple of 4. This is because the Tcl parser will first substitute **\$a + 2** for the variable **b**, then the **expr** command will evaluate the expression **\$a + 2\*4**.

Most expressions do not require a second round of substitutions. Either they are enclosed in braces or, if not, their variable and command substitutions yield numbers or strings that don't themselves require substitutions. However, because a few unbraced expressions need two rounds of substitutions, the bytecode compiler must emit additional instructions to handle this situation. The most expensive code is required for unbraced expressions that contain command substitutions. These expressions must be implemented by generating new code each time the expression is executed.

## 2.19 **fconfigure** - Set and Get Options on a Channel

### Name

**fconfigure** - Set and get options on a channel

### Synopsis

**fconfigure** *channelId*

**fconfigure** *channelId name*

**fconfigure** *channelId name value ?name value ...?*

### Description

The **fconfigure** command sets and retrieves options for channels. *ChannelId* identifies the channel for which to set or query an option. If no *name* or *value* arguments are supplied, the command returns a list containing alternating option names and values for the channel. If *name* is supplied but no *value* then the command returns the current value of the given option. If one or more pairs of *name* and *value* are supplied, the command sets each of the named options to the corresponding *value*; in this case the return value is an empty string.

The options described below are supported for all channels. In addition, each channel type may add options that only it supports. See the manual entry for the command that creates each type of channels for the options that that specific type of channel supports.

For example, see the manual entry for the **socket** command for its additional options.

#### **-blocking** *boolean*

The **-blocking** option determines whether I/O operations on the channel can cause the process to block indefinitely. The value of the option must be a proper boolean value. Channels are normally in blocking mode; if a channel is placed into nonblocking mode it will affect the operation of the **gets**, **read**, **puts**, **flush**, and **close** commands; see the documentation for those commands for details. For nonblocking mode to work correctly, the application must be using the Tcl event loop (e.g. by calling **Tcl\_DoOneEvent** or invoking the **vwait** command).

#### **-buffering** *newValue*

If *newValue* is **full** then the I/O system will buffer output until its internal buffer is full or until the **flush** command is invoked. If *newValue* is **line**, then the I/O system will automatically flush output for the channel whenever a newline character is output. If *newValue* is **none**, the I/O system will flush automatically after every output operation. The default is for **-buffering** to be set to **full** except for channels that connect to terminal-like devices; for these channels the initial setting is **line**.

#### **-buffersize** *newSize*

*Newvalue* must be an integer; its value is used to set the size of buffers, in bytes, subsequently allocated for this channel to store input or output. *Newvalue* must be between ten and one million, allowing buffers of ten to one million bytes in size.

#### **-eofchar** *char*

#### **-eofchar** {*inChar outChar*}

This option supports DOS file systems that use Control-z (\x1a) as an end of file marker. If *char* is not an empty string, then this character signals end of file when it is encountered during input. For output, the end of file character is output when the channel is closed. If *char* is the empty string, then there is no special end of file character marker. For read-write channels, a two-element list specifies the end of file

marker for input and output, respectively. As a convenience, when setting the end-of-file character for a read-write channel you can specify a single value that will apply to both reading and writing. When querying the end-of-file character of a read-write channel, a two-element list will always be returned. The default value for **-eofchar** is the empty string in all cases. In that case the **-eofchar** is Control-z (\x1a) for reading and the empty string for writing.

**-translation** *mode*

**-translation** {*inMode outMode*}

In Tcl scripts the end of a line is always represented using a single newline character (\n). However, in actual files and devices the end of a line may be represented differently on different platforms, or even for different devices on the same platform. For example, under UNIX newlines are used in files, whereas carriage-return-linefeed sequences are normally used in network connections. On input (i.e., with **gets** and **read**) the Tcl I/O system automatically translates the external end-of-line representation into newline characters. Upon output (i.e., with **puts**), the I/O system translates newlines to the external end-of-line representation. The default translation mode, **auto**, handles all the common cases automatically, but the **-translation** option provides explicit control over the end of line translations.

The value associated with **-translation** is a single item for read-only and write-only channels. The value is a two-element list for read-write channels; the read translation mode is the first element of the list, and the write translation mode is the second element. As a convenience, when setting the translation mode for a read-write channel you can specify a single value that will apply to both reading and writing. When querying the translation mode of a read-write channel, a two-element list will always be returned. The following values are currently supported:

#### **auto**

As the input translation mode, **auto** treats any of newline (**lf**), carriage return (**cr**), or carriage return followed by a newline (**crlf**) as the end of line representation. The end of line representation can even change from line-to-line, and all cases are translated to a newline. As the output translation mode, **auto** chooses a platform specific representation; for sockets on all platforms Tcl chooses **crlf**, for all Unix flavors, it chooses **lf**. The default setting for **-translation** is **auto** for both input and output.

#### **binary**

No end-of-line translations are performed. This is nearly identical to **lf** mode, except that in addition **binary** mode also sets the end of file character to the empty string, which disables it. See the description of **-eofchar** for more information.

#### **cr**

The end of a line in the underlying file or device is represented by a single carriage return character. As the input translation mode, **cr** mode converts carriage returns to newline characters. As the output translation mode, **cr** mode translates newline characters to carriage returns.

#### **crlf**

The end of a line in the underlying file or device is represented by a carriage return character followed by a linefeed character. As the input translation mode, **crlf** mode converts carriage-return-linefeed sequences to newline characters. As the output translation mode, **crlf** mode translates newline characters to carriage-return-linefeed sequences. This mode is typically used for network connections.

#### **lf**

The end of a line in the underlying file or device is represented by a single newline (linefeed) character. In this mode no translations occur during either input or output. This mode is typically used on UNIX platforms.

**See also**

close, flush, gets, puts, read, socket

## 2.20 **fcopy** - Copy Data From One Channel to Another

### Name

**fcopy** - Copy data from one channel to another.

### Synopsis

**fcopy** *inchan outchan* *?-size size?* *?-command callback?*

### Description

The **fcopy** command copies data from one I/O channel, *inchan* to another I/O channel, *outchan*. The **fcopy** command leverages the buffering in the Tcl I/O system to avoid extra copies and to avoid buffering too much data in main memory when copying large files to slow destinations like network sockets.

The **fcopy** command transfers data from *inchan* until end of file or *size* bytes have been transferred. If no **-size** argument is given, then the copy goes until end of file. All the data read from *inchan* is copied to *outchan*. Without the **-command** option, **fcopy** blocks until the copy is complete and returns the number of bytes written to *outchan*.

The **-command** argument makes **fcopy** work in the background. In this case it returns immediately and the *callback* is invoked later when the copy completes. The *callback* is called with one or two additional arguments that indicates how many bytes were written to *outchan*. If an error occurred during the background copy, the second argument is the error string associated with the error. With a background copy, it is not necessary to put *inchan* or *outchan* into non-blocking mode; the **fcopy** command takes care of that automatically. However, it is necessary to enter the event loop by using the **vwait** command.

You are not allowed to do other I/O operations with *inchan* or *outchan* during a background **fcopy**. If either *inchan* or *outchan* get closed while the copy is in progress, the current copy is stopped and the command *callback* is *not* made. If *inchan* is closed, then all data already queued for *outchan* is written out.

Note that *inchan* can become readable during a background copy. You should turn off any **fileevent** handlers during a background copy so those handlers do not interfere with the copy. Any I/O attempted by a **fileevent** handler will get a "channel busy" error.

**Fcopy** translates end-of-line sequences in *inchan* and *outchan* according to the **-translation** option for these channels. See the manual entry for **fconfigure** for details on the **-translation** option. The translations mean that the number of bytes read from *inchan* can be different than the number of bytes written to *outchan*. Only the number of bytes written to *outchan* is reported, either as the return value of a synchronous **fcopy** or as the argument to the *callback* for an asynchronous **fcopy**.

### Examples

This first example shows how the callback gets passed the number of bytes transferred. It also uses `vwait` to put the application into the event loop. Of course, this simplified example could be done without the command callback.

```

proc Cleanup {in out bytes {error {}}} {
    global total
    set total $bytes
    close $in
    close $out
    if {[string length $error] != 0} {
        # error occurred during the copy
    }
}
set in [open $file1]
set out [socket $server $port]
fcopy $in $out -command [list Cleanup $in $out]
vwait total

```

The second example copies in chunks and tests for end of file in the command callback

```

proc CopyMore {in out chunk bytes {error {}}} {
    global total done
    incr total $bytes
    if {[string length $error] != 0} || [eof $in] {
        set done $total
        close $in
        close $out
    } else {
        fcopy $in $out -command [list CopyMore $in $out $chunk] \
            -size $chunk
    }
}
set in [open $file1]
set out [socket $server $port]
set chunk 1024
set total 0
fcopy $in $out -command [list CopyMore $in $out $chunk] -size $chunk
vwait done

```

### See also

`eof`, `fconfigure`

## 2.21 file - Manipulate File Names and Attributes

### Name

**file** - Manipulate file names and attributes

### Synopsis

**file** *option name ?arg arg ...?*

### Description

This command provides several operations on a file's name or attributes. *Name* is the name of a file; if it starts with a tilde, then tilde substitution is done before executing the command (see the manual entry for **filename** for details). *Option* indicates what to do with the file name. Any unique abbreviation for *option* is acceptable. The valid options are:

#### **file atime** *name*

Returns a decimal string giving the time at which file *name* was last accessed. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its access time cannot be queried then an error is generated.

#### **file attributes** *name*

**file attributes** *name ?option?*

**file attributes** *name ?option value option value...?*

This subcommand returns or sets platform specific values associated with a file. The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Unix, **-group** gets or sets the group name for the file. A group id can be given to the command, but it returns a group name. **-owner** gets or sets the user name of the owner of the file. The command returns the owner name, but the numerical id can be passed when setting the owner. **-permissions** sets or retrieves the octal code that `chmod(1)` uses. This command does not support the symbolic attributes for `chmod(1)` at this time.

**file copy** *?-force? ?--? source target*

**file copy** *?-force? ?--? source ?source ...? targetDir*

The first form makes a copy of the file or directory *source* under the pathname *target*. If *target* is an existing directory, then the second form is used. The second form makes a copy inside *targetDir* of each *source* file listed. If a directory is specified as a *source*, then the contents of the directory will be recursively copied into *targetDir*.

Existing files will not be overwritten unless the **-force** option is specified. Trying to overwrite a non-empty directory, overwrite a directory with a file, or a file with a directory will all result in errors even if *-force* was specified. Arguments are processed in the order specified, halting at the first error, if any. A `--` marks the end of switches; the argument following the `--` will be treated as a *source* even if it starts with a `-`.

**file delete** *?-force? ?--? pathname ?pathname ... ?*

Removes the file or directory specified by each *pathname* argument. Non-empty directories will be removed only if the **-force** option is specified. Trying to delete a non-existent file is not considered an error. Trying to delete a read-only file will cause the file to be deleted, even if the **-force** flag is not specified. Arguments are processed in the order specified, halting at the first error, if any. A `--` marks the end of switches; the argument following the `--` will be treated as a *pathname* even if it starts with a `-`.

**file dirname** *name*

Returns a name comprised of all of the path components in *name* excluding the last element. If *name* is a relative file name and only contains one path element, then returns `..`. If *name* refers to a root directory, then the root directory is returned.

For example,

**file dirname** `c:/`

returns `c/`.

Note that tilde substitution will only be performed if it is necessary to complete the command.

For example,

**file dirname** `~/src/foo.c`

returns `~/src`, whereas

**file dirname** `~`

returns `/home` (or something similar).

**file executable** *name*

Returns **1** if file *name* is executable by the current user, **0** otherwise.

**file exists** *name*

Returns **1** if file *name* exists and the current user has search privileges for the directories leading to it, **0** otherwise.

**file extension** *name*

Returns all of the characters in *name* after and including the last dot in the last element of *name*. If there is no dot in the last element of *name* then returns the empty string.

**file isdirectory** *name*

Returns **1** if file *name* is a directory, **0** otherwise.

**file isfile** *name*

Returns **1** if file *name* is a regular file, **0** otherwise.

**file join** *name ?name ...?*

Takes one or more file names and combines them, using the correct path separator for the current platform. If a particular *name* is relative, then it will be joined to the previous file name argument. Otherwise, any earlier arguments will be discarded, and joining will proceed from the current argument.

For example,

**file join** `a b /foo bar`

returns `/foo/bar`.

Note that any of the names can contain separators, and that the result is always canonical for the current platform: `/` for Unix .

**file lstat** *name varName*

Same as **stat** option (see below) except uses the *lstat* kernel call instead of *stat*. This means that if *name* refers to a symbolic link the information returned in *varName* is for the link rather than the file it refers to. On systems that don't support symbolic links this option behaves exactly the same as the **stat** option.

**file mkdir** *dir ?dir ...?*

Creates each directory specified. For each pathname *dir* specified, this command will create all non-existing parent directories as well as *dir* itself. If an existing directory is specified, then no action is taken and no error is returned. Trying to overwrite an existing file with a directory will result in an error. Arguments are processed in the order specified, halting at the first error, if any.

**file mtime** *name*

Returns a decimal string giving the time at which file *name* was last modified. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its modified time cannot be queried then an error is generated.

**file nativename** *name*

Returns the platform-specific name of the file. This is useful if the filename is needed to pass to a platform-specific call.

**file owned** *name*

Returns **1** if file *name* is owned by the current user, **0** otherwise.

**file pathtype** *name*

Returns one of **absolute**, **relative**, **volumerelative**. If *name* refers to a specific file on a specific volume, the path type will be **absolute**. If *name* refers to a file relative to the current working directory, then the path type will be **relative**. If *name* refers to a file relative to the current working directory on a specified volume, or to a specific file on the current working volume, then the file type is **volumerelative**.

**file readable** *name*

Returns **1** if file *name* is readable by the current user, **0** otherwise.

**file readlink** *name*

Returns the value of the symbolic link given by *name* (i.e. the name of the file it points to). If *name* isn't a symbolic link or its value cannot be read, then an error is returned. On systems that don't support symbolic links this option is undefined.

**file rename** *?-force? ?--? source target***file rename** *?-force? ?--? source ?source ...? targetDir*

The first form takes the file or directory specified by pathname *source* and renames it to *target*, moving the file if the pathname *target* specifies a name in a different directory. If *target* is an existing directory, then the second form is used. The second form moves each *source* file or directory into the directory *targetDir*. Existing files will not be overwritten unless the **-force** option is specified. Trying to overwrite a non-empty directory, overwrite a directory with a file, or a file with a directory will all result in errors. Arguments are processed in the order specified, halting at the first error, if any. A **--** marks the end of switches; the argument following the **--** will be treated as a *source* even if it starts with a **-**.

**file rootname** *name*

Returns all of the characters in *name* up to but not including the last **`.`** character in the last component of name. If the last component of *name* doesn't contain a dot, then returns *name*.

**file size** *name*

Returns a decimal string giving the size of file *name* in bytes. If the file doesn't exist or its size cannot be queried then an error is generated.

**file split** *name*

Returns a list whose elements are the path components in *name*. The first element of the list will have the same path type as *name*. All other elements will be relative. Path separators will be discarded unless they are needed ensure that an element is unambiguously relative. For example, under Unix

**file split** */foo/~bar/baz*

returns **/ foo ./~bar baz** to ensure that later commands that use the third component do not attempt to perform tilde substitution.

**file stat** *name varName*

Invokes the **stat** kernel call on *name*, and uses the variable given by *varName* to hold information returned from the kernel call. *VarName* is treated as an array variable, and the following elements of that variable are set: **atime**, **ctime**, **dev**, **gid**, **ino**, **mode**, **mtime**, **nlink**, **size**, **type**, **uid**. Each element except **type** is a decimal string with the value of the corresponding field from the **stat** return structure; see the manual entry for **stat** for details on the meanings of the values. The **type** element gives the type of the file in the same form returned by the command **file type**. This command returns an empty string.

**file tail** *name*

Returns all of the characters in *name* after the last directory separator. If *name* contains no separators then returns *name*.

**file type** *name*

Returns a string giving the type of file *name*, which will be one of **file**, **directory**, **characterSpecial**, **blockSpecial**, **fifo**, **link**, or **socket**.

**file volume**

Returns the absolute paths to the volumes mounted on the system, as a proper Tcl list. On UNIX, the command will always return "/", since all filesystems are locally mounted.

**file writable** *name*

Returns **1** if file *name* is writable by the current user, **0** otherwise.

**Portability issues****Unix**

These commands always operate using the real user and group identifiers, not the effective ones.

## 2.22 **fileevent - Execute a Script When a Channel Becomes Readable or Writable**

### Name

**fileevent** - Execute a script when a channel becomes readable or writable

### Synopsis

**fileevent** *channelId* **readable** *?script?*

**fileevent** *channelId* **writable** *?script?*

### Description

This command is used to create *file event handlers*. A file event handler is a binding between a channel and a script, such that the script is evaluated whenever the channel becomes readable or writable. File event handlers are most commonly used to allow data to be received from another process on an event-driven basis, so that the receiver can continue to interact with the user while waiting for the data to arrive. If an application invokes **gets** or **read** on a blocking channel when there is no input data available, the process will block; until the input data arrives, it will not be able to service other events, so it will appear to the user to "freeze up". With **fileevent**, the process can tell when data is present and only invoke **gets** or **read** when they won't block.

The *channelId* argument to **fileevent** refers to an open channel, such as the return value from a previous **open** or **socket** command. If the *script* argument is specified, then **fileevent** creates a new event handler: *script* will be evaluated whenever the channel becomes readable or writable (depending on the second argument to **fileevent**). In this case **fileevent** returns an empty string. The **readable** and **writable** event handlers for a file are independent, and may be created and deleted separately. However, there may be at most one **readable** and one **writable** handler for a file at a given time in a given interpreter. If **fileevent** is called when the specified handler already exists in the invoking interpreter, the new script replaces the old one.

If the *script* argument is not specified, **fileevent** returns the current script for *channelId*, or an empty string if there is none. If the *script* argument is specified as an empty string then the event handler is deleted, so that no script will be invoked. A file event handler is also deleted automatically whenever its channel is closed or its interpreter is deleted.

A channel is considered to be readable if there is unread data available on the underlying device. A channel is also considered to be readable if there is unread data in an input buffer, except in the special case where the most recent attempt to read from the channel was a **gets** call that could not find a complete line in the input buffer. This feature allows a file to be read a line at a time in nonblocking mode using events. A channel is also considered to be readable if an end of file or error condition is present on the underlying file or device. It is important for *script* to check for these conditions and handle them appropriately; for example, if there is no special check for end of file, an infinite loop may occur where *script* reads no data, returns, and is immediately invoked again.

A channel is considered to be writable if at least one byte of data can be written to the underlying file or device without blocking, or if an error condition is present on the underlying file or device.

Event-driven I/O works best for channels that have been placed into nonblocking mode with the **fconfigure** command. In blocking mode, a **puts** command may block if you give it more data than the underlying file or device can accept, and a **gets** or **read** command will block if you attempt to read more data than is ready; no events will be processed while the commands block. In nonblocking mode **puts**, **read**, and **gets** never

block. See the documentation for the individual commands for information on how they handle blocking and nonblocking channels.

The script for a file event is executed at global level (outside the context of any Tcl procedure) in the interpreter in which the **fileevent** command was invoked. In addition, the file event handler is deleted if it ever returns an error; this is done in order to prevent infinite loops due to buggy handlers.

**See also**

**fconfigure, gets, puts, read**

## 2.23 flush - Flush Buffered Output for a Channel

### Name

**flush** - Flush buffered output for a channel

### Synopsis

**flush** *channelId*

### Description

Flushes any output that has been buffered for *channelId*. *ChannelId* must be a channel identifier such as returned by a previous **open** or **socket** command, and it must have been opened for writing. If the channel is in blocking mode the command does not return until all the buffered output has been flushed to the channel. If the channel is in nonblocking mode, the command may return before all buffered output has been flushed; the remainder will be flushed in the background as fast as the underlying file or device is able to absorb it.

### See also

**open**, **socket**

## 2.24 for - ``For'' Loop

### Name

**for** - ``For'' loop

### Synopsis

**for** *start test next body*

### Description

**For** is a looping command, similar in structure to the C **for** statement. The *start*, *next*, and *body* arguments must be Tcl command strings, and *test* is an expression string. The **for** command first invokes the Tcl interpreter to execute *start*. Then it repeatedly evaluates *test* as an expression; if the result is non-zero it invokes the Tcl interpreter on *body*, then invokes the Tcl interpreter on *next*, then repeats the loop. The command terminates when *test* evaluates to 0. If a **continue** command is invoked within *body* then any remaining commands in the current execution of *body* are skipped; processing continues by invoking the Tcl interpreter on *next*, then evaluating *test*, and so on. If a **break** command is invoked within *body* or *next*, then the **for** command will return immediately. The operation of **break** and **continue** are similar to the corresponding statements in C. **For** returns an empty string.

---

### NOTE

*test* should almost always be enclosed in braces. If not, variable substitutions will be made before the **for** command starts executing, which means that variable changes made by the loop body will not be considered in the expression. This is likely to result in an infinite loop. If *test* is enclosed in braces, variable substitutions are delayed until the expression is evaluated (before each loop iteration), so changes in the variables will be visible.

---

For an example, try the following script with and without the braces around  $\$x < 10$ :

```
for {set x 0} { $\$x < 10$ } {incr x} {  
    puts "x is  $\$x$ "  
}
```

## 2.25 **foreach** - Iterate Over All Elements in One or More Lists

### Name

**foreach** - Iterate over all elements in one or more lists

### Synopsis

**foreach** *varname list body*

**foreach** *varlist1 list1 ?varlist2 list2 ...? body*

### Description

The **foreach** command implements a loop where the loop variable(s) take on values from one or more lists. In the simplest case there is one loop variable, *varname*, and one list, *list*, that is a list of values to assign to *varname*. The *body* argument is a Tcl script. For each element of *list* (in order from first to last), **foreach** assigns the contents of the element to *varname* as if the **lindex** command had been used to extract the element, then calls the Tcl interpreter to execute *body*.

In the general case there can be more than one value list (e.g., *list1* and *list2*), and each value list can be associated with a list of loop variables (e.g., *varlist1* and *varlist2*). During each iteration of the loop the variables of each *varlist* are assigned consecutive values from the corresponding *list*. Values in each *list* are used in order from first to last, and each value is used exactly once. The total number of loop iterations is large enough to use up all the values from all the value lists. If a value list does not contain enough elements for each of its loop variables in each iteration, empty values are used for the missing elements.

The **break** and **continue** statements may be invoked inside *body*, with the same effect as in the **for** command. **foreach** returns an empty string.

### Examples

The following loop uses *i* and *j* as loop variables to iterate over pairs of elements of a single list.

```
set x {}
foreach {i j} {a b c d e f} {
    lappend x $j $i
}
# The value of x is "b a d c f e"
# There are 3 iterations of the loop.
```

The next loop uses *i* and *j* to iterate over two lists in parallel.

```
set x {}
foreach i {a b c} j {d e f g} {
    lappend x $i $j
}
# The value of x is "a d b e c f g"
# There are 4 iterations of the loop.
```

The two forms are combined in the following example.

```
set x {}
foreach i {a b c} {j k} {d e f g} {
    lappend x $i $j $k
}
# The value of x is "a d e b f g c f g"
# There are 3 iterations of the loop.
```

## 2.26 **format - Format a String in the Style of sprintf**

### Name

**format** - Format a string in the style of sprintf

### Synopsis

**format** *formatString* ?*arg arg ...*?

### Introduction

This command generates a formatted string in the same way as the ANSI C **sprintf** procedure (it uses **sprintf** in its implementation). *FormatString* indicates how to format the result, using % conversion specifiers as in **sprintf**, and the additional arguments, if any, provide values to be substituted into the result. The return value from **format** is the formatted string.

### Details on formatting

The command operates by scanning *formatString* from left to right. Each character from the format string is appended to the result string unless it is a percent sign. If the character is a % then it is not copied to the result string. Instead, the characters following the % character are treated as a conversion specifier. The conversion specifier controls the conversion of the next successive *arg* to a particular format and the result is appended to the result string in place of the conversion specifier.

If there are multiple conversion specifiers in the format string, then each one controls the conversion of one additional *arg*. The **format** command must be given enough *args* to meet the needs of all of the conversion specifiers in *formatString*.

Each conversion specifier may contain up to six different parts: an XPG3 position specifier, a set of flags, a minimum field width, a precision, a length modifier, and a conversion character. Any of these fields may be omitted except for the conversion character. The fields that are present must appear in the order given above. The paragraphs below discuss each of these fields in turn.

If the % is followed by a decimal number and a \$, as in ``%2\$d'', then the value to convert is not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where 1 corresponds to the first *arg*. If the conversion specifier requires multiple arguments because of \* characters in the specifier then successive arguments are used, starting with the argument given by the number. This follows the XPG3 conventions for positional specifiers. If there are any positional specifiers in *formatString* then all of the specifiers must be positional.

The **second portion of a conversion** specifier may contain any of the following flag characters, in any order:

- ..... Specifies that the converted argument should be left-justified in its field (numbers are normally right-justified with leading spaces if needed).
- +..... Specifies that a number should always be printed with a sign, even if positive.
- space*..... Specifies that a space should be added to the beginning of the number if the first character isn't a sign.
- 0** ..... Specifies that the number should be padded on the left with zeroes instead of spaces.
- #**..... Requests an alternate output form. For **o** and **O** conversions it guarantees that the first digit is always **0**. For **x** or **X** conversions, **0x** or **0X** (respectively) will be added to the beginning of the result unless it is zero. For all floating-

point conversions (**e**, **E**, **f**, **g**, and **G**) it guarantees that the result always has a decimal point. For **g** and **G** conversions it specifies that trailing zeroes should not be removed.

The **third portion of a conversion** specifier is a number giving a minimum field width for this conversion. It is typically used to make columns line up in tabular printouts. If the converted argument contains fewer characters than the minimum field width then it will be padded so that it is as wide as the minimum field width. Padding normally occurs by adding extra spaces on the left of the converted argument, but the **0** and **-** flags may be used to specify padding with zeroes on the left or with spaces on the right, respectively. If the minimum field width is specified as **\*** rather than a number, then the next argument to the **format** command determines the minimum field width; it must be a numeric string.

The fourth portion of a conversion specifier is a precision, which consists of a period followed by a number. The number is used in different ways for different conversions.

For **e**, **E**, and **f** conversions it specifies the number of digits to appear to the right of the decimal point.

For **g** and **G** conversions it specifies the total number of digits to appear, including those on both sides of the decimal point (however, trailing zeroes after the decimal point will still be omitted unless the **#** flag has been specified). For integer conversions, it specifies a minimum number of digits to print (leading zeroes will be added if necessary). For **s** conversions it specifies the maximum number of characters to be printed; if the string is longer than this then the trailing characters will be dropped. If the precision is specified with **\*** rather than a number then the next argument to the **format** command determines the precision; it must be a numeric string.

The fifth part of a conversion specifier is a length modifier, which must be **h** or **l**. If it is **h** it specifies that the numeric value should be truncated to a 16-bit value before converting. This option is rarely useful. The **l** modifier is ignored.

The last thing in a conversion specifier is an alphabetic character that determines what kind of conversion to perform. The following conversion characters are currently supported:

- d** ..... Convert integer to signed decimal string.
- u** ..... Convert integer to unsigned decimal string.
- i** ..... Convert integer to signed decimal string; the integer may either be in decimal, in octal (with a leading **0**) or in hexadecimal (with a leading **0x**).
- o** ..... Convert integer to unsigned octal string.
- x** or **X** ..... Convert integer to unsigned hexadecimal string, using digits ``0123456789abcdef`` for **x** and ``0123456789ABCDEF`` for **X**.
- c** ..... Convert integer to the 8-bit character it represents.
- s** ..... No conversion; just insert string.
- f** ..... Convert floating-point number to signed decimal string of the form `xx.yyy`, where the number of *y*'s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output.
- e** or **e** ..... Convert floating-point number to scientific notation in the form `x.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. If the **E** form is used then **E** is printed instead of **e**.
- g** or **G** ..... If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for **%e** or **%E**. Otherwise convert as for **%f**. Trailing zeroes and a trailing decimal point are omitted.
- %** ..... No conversion: just insert **%**.

For the numerical conversions the argument being converted must be an integer or floating-point string; `format` converts the argument to binary and then converts it back to a string according to the conversion specifier.

### **Differences from `ansi sprintf`**

The behavior of the `format` command is the same as the ANSI C `sprintf` procedure except for the following differences:

[1]

`%p` and `%n` specifiers are not currently supported.

[2]

For `%c` conversions the argument must be a decimal string, which will then be converted to the corresponding character value.

[3]

The `l` modifier is ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the `l` modifier were present (i.e. type `double` is used for the internal representation). If the `h` modifier is specified then integer values are truncated to `short` before conversion.

## 2.27 gets - Read a Line from a Channel

### Name

**gets** - Read a line from a channel

### Synopsis

**gets** *channelId* ?*varName*?

### Description

This command reads the next line from *channelId*, returns everything in the line up to (but not including) the end-of-line character(s), and discards the end-of-line character(s). If *varName* is omitted the line is returned as the result of the command. If *varName* is specified then the line is placed in the variable by that name and the return value is a count of the number of characters returned.

If end of file occurs while scanning for an end of line, the command returns whatever input is available up to the end of file. If *channelId* is in nonblocking mode and there is not a full line of input available, the command returns an empty string and does not consume any input. If *varName* is specified and an empty string is returned in *varName* because of end-of-file or because of insufficient data in nonblocking mode, then the return count is -1.

Note that if *varName* is not specified then the end-of-file and no-full-line-available cases can produce the same results as if there were an input line consisting only of the end-of-line character(s). The **eof** command can be used to distinguish these three cases.

### See also

**eof**

## 2.28 **glob - Return Names of Files that Match Patterns**

### Name

**glob** - Return names of files that match patterns

### Synopsis

**glob** *?switches? pattern ?pattern ...?*

### Description

This command performs file name "globbing" in a fashion similar to the csh shell. It returns a list of the files whose names match any of the *pattern* arguments.

If the initial arguments to **glob** start with - then they are treated as switches. The following switches are currently supported:

#### **-nocomplain**

Allows an empty list to be returned without error; without this switch an error is returned if the result list would be empty.

--

Marks the end of switches. The argument following this one will be treated as a *pattern* even if it starts with a -.

The *pattern* arguments may contain any of the following special characters:

?

Matches any single character.

\*

Matches any sequence of zero or more characters.

[*chars*]

Matches any single character in *chars*. If *chars* contains a sequence of the form *a-b* then any character between *a* and *b* (inclusive) will match.

\*x*

Matches the character *x*.

{*a,b,...*}

Matches any of the strings *a*, *b*, etc.

As with csh, a "." at the beginning of a file's name or just after a "/" must be matched explicitly or with a {} construct. In addition, all "/" characters must be matched explicitly.

If the first character in a *pattern* is "~" then it refers to the home directory for the user whose name follows the "~". If the "~" is followed immediately by "/" then the value of the HOME environment variable is used.

The **glob** command differs from csh globbing in two ways. First, it does not sort its result list (use the **lsort** command if you want the list sorted). Second, **glob** only returns the names of files that actually exist; in csh no check for existence is made unless a pattern contains a ?, \*, or [] construct.

### Portability issues

Unlike other Tcl commands that will accept both network and native style names (see the **filename** manual entry for details on how native and network names are specified), the **glob** command only accepts native names.

## 2.29 **global** - Access Global Variables

### Name

**global** - Access global variables

### Synopsis

**global** *varname* ?*varname* ...?

### Description

This command is ignored unless a Tcl procedure is being interpreted. If so then it declares the given *varname*'s to be global variables rather than local ones. Global variables are variables in the global namespace. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varnames* will refer to the global variable by the same name.

Example:

```
proc muet_proc {} {  
    # Set local variable  
    set mavar 4  
    puts "La valeur de la variable locale est $mavar"  
  
    # Access to global variable  
    global variableGlob  
    puts "La valeur de la variable globale est $variableGlob"  
}  
  
# Set global variable  
set variableGlob 15  
  
# Call procedure  
muet_proc
```

### See also

namespace, variable

## 2.30 if - Execute Scripts Conditionally

### Name

**if** - Execute scripts conditionally

### Synopsis

**if** *expr1* **?then?** *body1* **elseif** *expr2* **?then?** *body2* **elseif** ... **?else?** *?bodyN?*

### Description

The *if* command evaluates *expr1* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter.

Otherwise *expr2* is evaluated as an expression and if it is true then **body2** is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read.

There may be any number of **elseif** clauses, including zero. *bodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

Example of multiple conditions:

```
if {$x == 0 && $y == 0} {  
    ...  
}
```

## 2.31 **incr - Increment the Value of a Variable**

### **Name**

**incr** - Increment the value of a variable

### **Synopsis**

**incr** *varName* *?increment?*

### **Description**

Increments the value stored in the variable whose name is *varName*. The value of the variable must be an integer. If *increment* is supplied then its value (which must be an integer) is added to the value of variable *varName*; otherwise 1 is added to *varName*.

The new value is stored as a decimal string in variable *varName* and also returned as result.

## 2.32 info - Return Information About the State of the Tcl Interpreter

### Name

**info** - Return information about the state of the Tcl interpreter

### Synopsis

**info** *option ?arg arg ...?*

### Description

This command provides information about various internals of the Tcl interpreter. The legal *option*'s (which may be abbreviated) are:

#### **info args** *procname*

Returns a list containing the names of the arguments to procedure *procname*, in order. *Procname* must be the name of a Tcl command procedure.

#### **info body** *procname*

Returns the body of procedure *procname*. *Procname* must be the name of a Tcl command procedure.

#### **info cmdcount**

Returns a count of the total number of commands that have been invoked in this interpreter.

#### **info commands** *?pattern?*

If *pattern* isn't specified, returns a list of names of all the Tcl commands in the current namespace, including both the built-in commands written in C and the command procedures defined using the [proc](#) command. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**. *pattern* can be a qualified name like **Foo::print\***. That is, it may specify a particular namespace using a sequence of namespace names separated by **::**s, and may have pattern matching special characters at the end to specify a set of commands in that namespace. If *pattern* is a qualified name, the resulting list of command names has each one qualified with the name of the specified namespace.

#### **info complete** *command*

Returns 1 if *command* is a complete Tcl command in the sense of having no unclosed quotes, braces, brackets or array element names, If the command doesn't appear to be complete then 0 is returned. This command is typically used in line-oriented input environments to allow users to type in commands that span multiple lines; if the command isn't complete, the script can delay evaluating it until additional lines have been typed to complete the command.

#### **info default** *procname arg varname*

*Procname* must be the name of a Tcl command procedure and *arg* must be the name of an argument to that procedure. If *arg* doesn't have a default value then the command returns 0. Otherwise it returns 1 and places the default value of *arg* into variable *varname*.

#### **info exists** *varName*

Returns 1 if the variable named *varName* exists in the current context (either as a global or local variable), returns 0 otherwise.

#### **info globals** *?pattern?*

If *pattern* isn't specified, returns a list of all the names of currently-defined global variables. Global variables are variables in the global namespace. If *pattern* is specified,

only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

**info hostname**

Returns the name of the computer on which this invocation is being executed.

**info level ?number?**

If *number* is not specified, this command returns a number giving the stack level of the invoking procedure, or 0 if the command is invoked at top-level. If *number* is specified, then the result is a list consisting of the name and arguments for the procedure call at level *number* on the stack. If *number* is positive then it selects a particular stack level (1 refers to the top-most active procedure, 2 to the procedure it called, and so on); otherwise it gives a level relative to the current level (0 refers to the current procedure, -1 to its caller, and so on). See the [uplevel](#) command for more information on what stack levels mean.

**info library**

Returns the name of the library directory in which standard Tcl scripts are stored. This is actually the value of the **tcl\_library** variable and may be changed by setting **tcl\_library**. See the [tclvars](#) manual entry for more information.

**info loaded ?interp?**

Returns a list describing all of the packages that have been loaded into *interp* with the [load](#) command. Each list element is a sub-list with two elements consisting of the name of the file from which the package was loaded and the name of the package. For statically-loaded packages the file name will be an empty string. If *interp* is omitted then information is returned for all packages loaded in any interpreter in the process. To get a list of just the packages in the current interpreter, specify an empty string for the *interp* argument.

**info locals ?pattern?**

If *pattern* isn't specified, returns a list of all the names of currently-defined local variables, including arguments to the current procedure, if any. Variables defined with the **global** and **upvar** commands will not be returned. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

**info nameofexecutable**

Returns the full path name of the binary file from which the application was invoked. If Tcl was unable to identify the file, then an empty string is returned.

**info patchlevel**

Returns the value of the global variable **tcl\_patchLevel**; see the **tclvars** manual entry for more information.

**info procs ?pattern?**

If *pattern* isn't specified, returns a list of all the names of Tcl command procedures in the current namespace. If *pattern* is specified, only those procedure names in the current namespace matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

**info script**

If a Tcl script file is currently being evaluated (i.e. there is a call to **Tcl\_EvalFile** active or there is an active invocation of the **source** command), then this command returns the name of the innermost file being processed. Otherwise the command returns an empty string.

**info sharedlibextension**

Returns the extension used on this platform for the names of files containing shared libraries (for example, **.so** under Solaris). If shared libraries aren't supported on this platform then an empty string is returned.

**info tclversion**

Returns the value of the global variable **tcl\_version**; see the **tclvars** manual entry for more information.

**info vars ?*pattern*?**

If *pattern* isn't specified, returns a list of all the names of currently-visible variables. This includes locals and currently-visible globals. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**. *pattern* can be a qualified name like **Foo::option\***. That is, it may specify a particular namespace using a sequence of namespace names separated by **::**, and may have pattern matching special characters at the end to specify a set of variables in that namespace. If *pattern* is a qualified name, the resulting list of variable names has each matching namespace variable qualified with the name of its namespace.

## 2.33 **join - Create a String by Joining Together List Elements**

### Name

**join** - Create a string by joining together list elements

### Synopsis

**join** *list* ?*joinString*?

### Description

The *list* argument must be a valid Tcl list. This command returns the string formed by joining all of the elements of *list* together with *joinString* separating each adjacent pair of elements. The *joinString* argument defaults to a space character.

For example:

```
set maliste "Mercure Venus Mars"  
puts $maliste  
set str [ join $maliste ";" ]  
puts $str
```

=> *Mercure;Venus;Mars*

## 2.34 **lappend** - Append List Elements Onto a Variable

### Name

**lappend** - Append list elements onto a variable

### Synopsis

**lappend** *varName* ?*value value value ...?*

### Description

This command treats the variable given by *varName* as a list and appends each of the *value* arguments to that list as a separate element, with spaces between elements. If *varName* doesn't exist, it is created as a list with elements given by the *value* arguments. **Lappend** is similar to **append** except that the *values* are appended as list elements rather than raw text. This command provides a relatively efficient way to build up large lists.

For example, ``**lappend a \$b**'' is much more efficient than ``**set a [concat \$a [list \$b]]**'' when *\$a* is long.

For example:

```
set maliste "Mercure Venus Mars"
puts $maliste
lappend maliste Jupiter
puts $maliste
```

=> *Mercure Venus Mars*

=> *Mercure Venus Mars Jupiter*

## 2.35 **lindex** - Retrieve an Element From a List

### Name

**lindex** - Retrieve an element from a list

### Synopsis

**lindex** *list index*

### Description

This command treats *list* as a Tcl list and returns the *index*'th element from it (0 refers to the first element of the list). In extracting the element, *lindex* observes the same rules concerning braces and quotes and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If *index* is negative or greater than or equal to the number of elements in *value*, then an empty string is returned. If *index* has the value **end**, it refers to the last element in the list.

Example to use lindex:

```
set simple_list "John Joe Mary Susan"
puts [lindex $simple_list 0]
puts [lindex $simple_list 2]
=> John
=> Mary
```

## 2.36 `linsert` - Insert Elements Into a List

### Name

`linsert` - Insert elements into a list

### Synopsis

`linsert` *list index element ?element element ...?*

### Description

This command produces a new list from *list* by inserting all of the *element* arguments just before the *index*th element of *list*. Each *element* argument will become a separate element of the new list. If *index* is less than or equal to zero, then the new elements are inserted at the beginning of the list. If *index* has the value **end**, or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

For example:

```
set maliste "Mercure Venus Mars"
puts $maliste
set maliste [ linsert $maliste 2 Terre ]
puts $maliste
=> Mercure Venus Mars
=> Mercure Venus Terre Mars
```

## 2.37 list - Create a List

### Name

**list** - Create a list

### Synopsis

**list** ?*arg arg ...?*

### Description

This command returns a list comprised of all the *args*, or an empty string if no *args* are specified. Braces and backslashes get added as necessary, so that the **index** command may be used on the result to re-extract the original arguments, and also so that **eval** may be used to execute the resulting list, with *arg1* comprising the command's name and the other *args* comprising its arguments. **List** produces slightly different results than **concat**: **concat** removes one level of grouping before forming the list, while **list** works directly from the original arguments.

For example, the command

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

will return: **a b {c d e} {f {g h}}**

while **concat** with the same arguments will return: **a b c d e f {g h}**

For example:

```
set simple_liste2 "Michael Samuel Sophie Stéphanie"  
set groupe_liste [list $simple_liste $simple_liste2]  
puts $groupe_liste
```

=> *{John Joe Mary Susan} {Michael Samuel Sophie Stéphanie}*

## 2.38 **llength** - Count the Number of Elements in a List

### Name

**llength** - Count the number of elements in a list

### Synopsis

**llength** *list*

### Description

Treats *list* as a list and returns a decimal string giving the number of elements in it.

For example:

```
set simple_liste2 "Michael Samuel Sophie Stéphanie"
set groupe_liste [list $simple_liste $simple_liste2]
puts [llength $groupe_liste]
=> 2
```

## 2.39 `lrange` - Return One or More Adjacent Elements From a List

### Name

`lrange` - Return one or more adjacent elements from a list

### Synopsis

`lrange` *list first last*

### Description

*List* must be a valid Tcl list. This command will return a new list consisting of elements *first* through *last*, inclusive.

*First* or *last* may be **end** (or any abbreviation of it) to refer to the last element of the list.

If *first* is less than zero, it is treated as if it were zero.

If *last* is greater than or equal to the number of elements in the list, then it is treated as if it were **end**.

If *first* is greater than *last* then an empty string is returned.

---

### NOTE

**``lrange *list first first*'' does not always produce the same result as ``lindex *list first*'' (although it often does for simple fields that aren't enclosed in braces); it does, however, produce exactly the same results as ``list [lindex *list first*]''**

---

For example:

```
set maliste "Mercure Venus Terre Mars Jupiter"
set listeRange [ lrange $maliste 0 2]
puts $listeRange
```

=> *Mercure Venus Terre*

## 2.40 **lreplace** - Replace Elements in a List With New Elements

### Name

**lreplace** - Replace elements in a list with new elements

### Synopsis

**lreplace** *list first last ?element element ...?*

### Description

**lreplace** returns a new list formed by replacing one or more elements of *list* with the *element* arguments. *First* gives the index in *list* of the first element to be replaced (0 refers to the first element). If *first* is less than zero then it refers to the first element of *list*; the element indicated by *first* must exist in the list. *Last* gives the index in *list* of the last element to be replaced. If *last* is less than *first* then no elements are deleted; the new elements are simply inserted before *first*. *First* or *last* may be **end** (or any abbreviation of it) to refer to the last element of the list. The *element* arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each *element* argument will become a separate element of the list. If no *element* arguments are specified, then the elements between *first* and *last* are simply deleted.

```
set maliste "Mercure Venus Terre Mars Jupiter"  
set newliste [ lreplace $maliste 0 0 Lune]  
puts $newliste
```

=> *Lune Venus Terre Mars Jupiter*

## 2.41 lsearch - See if a List Contains a Particular Element

### Name

**lsearch** - See if a list contains a particular element

### Synopsis

**lsearch** *?mode?* *list pattern*

### Description

This command searches the elements of *list* to see if one of them matches *pattern*. If so, the command returns the index of the first matching element. If not, the command returns **-1**. The *mode* argument indicates how the elements of the list are to be matched against *pattern* and it must have one of the following values:

#### **-exact**

The list element must contain exactly the same string as *pattern*.

#### **-glob**

*Pattern* is a glob-style pattern which is matched against each list element using the same rules as the **string match** command.

#### **-regexp**

*Pattern* is treated as a regular expression and matched against each list element using the same rules.

If *mode* is omitted then it defaults to **-glob**.

For example:

```
set maliste "Mercure Venus Terre Mars Jupiter"
set index [ lsearch -exact $maliste Terre ]
puts $index
=> 2
```

## 2.42 **lsort** - Sort the Elements of a List

### Name

**lsort** - Sort the elements of a list

### Synopsis

**lsort** *?options?* *list*

### Description

This command sorts the elements of *list*, returning a new list in sorted order. By default ASCII sorting is used with the result returned in increasing order. However, any of the following options may be specified before *list* to control the sorting process (unique abbreviations are accepted):

- ascii** ..... Use string comparison with ASCII collation order. This is the default.
- dictionary** ..... Use dictionary-style comparison. This is the same as **-ascii** except (a) case is ignored except as a tie-breaker and (b) if two strings contain embedded numbers, the numbers compare as integers, not characters. For example, in **-dictionary** mode, **bigBoy** sorts between **bigbang** and **bigboy**, and **x10y** sorts between **x9y** and **x11y**.
- integer** ..... Convert list elements to integers and use integer comparison.
- real** ..... Convert list elements to floating-point values and use floating comparison.
- command** *command* ..... Use *command* as a comparison command. To compare two elements, evaluate a Tcl script consisting of *command* with the two elements appended as additional arguments. The script should return an integer less than, equal to, or greater than zero if the first element is to be considered less than, equal to, or greater than the second, respectively.
- increasing** ..... Sort the list in increasing order ("smallest" items first). This is the default.
- decreasing** ..... Sort the list in decreasing order ("largest" items first).
- index** *index* ..... If this option is specified, each of the elements of *list* must itself be a proper Tcl sublist. Instead of sorting based on whole sublists, **lsort** will extract the *index*'th element from each sublist and sort based on the given element. The keyword **end** is allowed for the *index* to sort on the last sublist element.

For example,

```
lsort -integer -index 1 {{First 24} {Second 18} {Third 30}}
```

returns **{Second 18} {First 24} {Third 30}**. This option is much more efficient than using **-command** to achieve the same effect.

Other example:

```
set maliste "Mercury Venus Terre Mars Jupiter"
set newliste [ lsort -dictionary $maliste ]
puts $newliste
```

=> *Jupiter Mars Mercury Terre Venus*

## 2.43 namespace - Create and Manipulate Contexts for Commands and Variables

### Name

**namespace** - create and manipulate contexts for commands and variables

### Synopsis

**namespace** *?option? ?arg ...?*

### Description

The **namespace** command lets you create, access, and destroy separate contexts for commands and variables. See the section **WHAT IS A NAMESPACE?** below for a brief overview of namespaces. The legal *option*'s are listed below. Note that you can abbreviate the *option*'s.

**namespace children** *?namespace? ?pattern?*

Returns a list of all child namespaces that belong to the namespace *namespace*. If *namespace* is not specified, then the children are returned for the current namespace. This command returns fully-qualified names, which start with `::`. If the optional *pattern* is given, then this command returns only the names that match the glob-style pattern. The actual pattern used is determined as follows: a pattern that starts with `::` is used directly, otherwise the namespace *namespace* (or the fully-qualified name of the current namespace) is prepended onto the the pattern.

**namespace code** *script*

Captures the current namespace context for later execution of the script *script*. It returns a new script in which *script* has been wrapped in a **namespace code** command. The new script has two important properties. First, it can be evaluated in any namespace and will cause *script* to be evaluated in the current namespace (the one where the **namespace code** command was invoked). Second, additional arguments can be appended to the resulting script and they will be passed to *script* as additional arguments.

For example, suppose the command **set script [namespace code {foo bar}]** is invoked in namespace `::a::b`. Then **eval "\$script x y"** can be executed in any namespace (assuming the value of **script** has been passed in properly) and will have the same effect as the command **namespace eval ::a::b {foo bar x y}**. This command is needed because extensions like Tk normally execute callback scripts in the global namespace. A scoped command captures a command together with its namespace context in a way that allows it to be executed properly later. See the section **SCOPED VALUES** for some examples of how this is used to create callback scripts.

**namespace current**

Returns the fully-qualified name for the current namespace. The actual name of the global namespace is ```` (i.e., an empty string), but this command returns `::` for the global namespace as a convenience to programmers.

**namespace delete** *?namespace namespace ...?*

Each namespace *namespace* is deleted and all variables, procedures, and child namespaces contained in the namespace are deleted.

If a procedure is currently executing inside the namespace, the namespace will be kept alive until the procedure returns; however, the namespace is marked to prevent other code from looking it up by name.

If a namespace doesn't exist, this command returns an error. If no namespace names are given, this command does nothing.

**namespace eval** *namespace arg ?arg ...?*

Activates a namespace called *namespace* and evaluates some code in that context. If the namespace does not already exist, it is created. If more than one *arg* argument is specified, the arguments are concatenated together with a space between each one in the same fashion as the **eval** command, and the result is evaluated.

If *namespace* has leading namespace qualifiers and any leading namespaces do not exist, they are automatically created.

**namespace export** *?-clear? ?pattern pattern ...?*

Specifies which commands are exported from a namespace. The exported commands are those that can be later imported into another namespace using a **namespace import** command. Both commands defined in a namespace and commands the namespace has previously imported can be exported by a namespace. The commands do not have to be defined at the time the **namespace export** command is executed. Each *pattern* may contain glob-style special characters, but it may not include any namespace qualifiers. That is, the pattern can only specify commands in the current (exporting) namespace. Each *pattern* is appended onto the namespace's list of export patterns. If the **-clear** flag is given, the namespace's export pattern list is reset to empty before any *pattern* arguments are appended. If no *patterns* are given and the **-clear** flag isn't given, this command returns the namespace's current export list.

**namespace forget** *?pattern pattern ...?*

Removes previously imported commands from a namespace. Each *pattern* is a qualified name such as **foo::x** or **a::b::p\***. Qualified names contain **::s** and qualify a name with the name of one or more namespaces. Each *pattern* is qualified with the name of an exporting namespace and may have glob-style special characters in the command name at the end of the qualified name. Glob characters may not appear in a namespace name. This command first finds the matching exported commands. It then checks whether any of those those commands were previously imported by the current namespace. If so, this command deletes the corresponding imported commands. In effect, this un-does the action of a **namespace import** command.

**namespace import** *?-force? ?pattern pattern ...?*

Imports commands into a namespace. Each *pattern* is a qualified name like **foo::x** or **a::p\***. That is, it includes the name of an exporting namespace and may have glob-style special characters in the command name at the end of the qualified name. Glob characters may not appear in a namespace name. All the commands that match a *pattern* string and which are currently exported from their namespace are added to the current namespace. This is done by creating a new command in the current namespace that points to the exported command in its original namespace; when the new imported command is called, it invokes the exported command. This command normally returns an error if an imported command conflicts with an existing command. However, if the **-force** option is given, imported commands will silently replace existing commands. The **namespace import** command has snapshot semantics: that is, only requested commands that are currently defined in the exporting namespace are imported. In other words, you can import only the commands that are in a namespace at the time when the **namespace import** command is executed. If another command is defined and exported in this namespace later on, it will not be imported.

**namespace inscope** *namespace arg ?arg ...?*

Executes a script in the context of a particular namespace. This command is not expected to be used directly by programmers; calls to it are generated implicitly when applications use **namespace code** commands to create callback scripts that the applications then register with, e.g., Tk widgets. The **namespace inscope** command is much like the **namespace eval** command except that it has **lappend** semantics and the namespace must already exist. It treats the first argument as a list, and appends any arguments after the first onto the end as proper list elements. **namespace inscope ::foo**

`a x y z` is equivalent to `namespace eval ::foo [concat a [list x y z]]`. This **lappend** semantics is important because many callback scripts are actually prefixes.

#### **namespace origin** *command*

Returns the fully-qualified name of the original command to which the imported command *command* refers. When a command is imported into a namespace, a new command is created in that namespace that points to the actual command in the exporting namespace. If a command is imported into a sequence of namespaces *a, b, ..., n* where each successive namespace just imports the command from the previous namespace, this command returns the fully-qualified name of the original command in the first namespace, *a*. If *command* does not refer to an imported command, the command's own fully-qualified name is returned.

#### **namespace parent** *?namespace?*

Returns the fully-qualified name of the parent namespace for namespace *namespace*. If *namespace* is not specified, the fully-qualified name of the current namespace's parent is returned.

#### **namespace qualifiers** *string*

Returns any leading namespace qualifiers for *string*. Qualifiers are namespace names separated by `::`s. For the *string* `::foo::bar::x`, this command returns `::foo::bar`, and for `::` it returns ```` (an empty string). This command is the complement of the **namespace tail** command. Note that it does not check whether the namespace names are, in fact, the names of currently defined namespaces.

#### **namespace tail** *string*

Returns the simple name at the end of a qualified string. Qualifiers are namespace names separated by `::`s. For the *string* `::foo::bar::x`, this command returns `x`, and for `::` it returns ```` (an empty string). This command is the complement of the **namespace qualifiers** command. It does not check whether the namespace names are, in fact, the names of currently defined namespaces.

#### **namespace which** *?-command? ?-variable? name*

Looks up *name* as either a command or variable and returns its fully-qualified name. For example, if *name* does not exist in the current namespace but does exist in the global namespace, this command returns a fully-qualified name in the global namespace. If the command or variable does not exist, this command returns an empty string. If no flag is given, *name* is treated as a command name. See the section **NAME RESOLUTION** below for an explanation of the rules regarding name resolution.

### **What is a namespace?**

A namespace is a collection of commands and variables. It encapsulates the commands and variables to ensure that they won't interfere with the commands and variables of other namespaces. Tcl has always had one such collection, which we refer to as the *global namespace*. The global namespace holds all global variables and commands. The **namespace eval** command lets you create new namespaces.

For example,

```
namespace eval Counter {
    namespace export Bump
    variable num 0

    proc Bump {} {
        variable num
        incr num
    }
}
```

creates a new namespace containing the variable **num** and the procedure **Bump**. The commands and variables in this namespace are separate from other commands and variables in the same program. If there is a command named **Bump** in the global namespace, for example, it will be different from the command **Bump** in the **Counter** namespace.

Namespace variables resemble global variables in Tcl. They exist outside of the procedures in a namespace but can be accessed in a procedure via the **variable** command, as shown in the example above.

Namespaces are dynamic. You can add and delete commands and variables at any time, so you can build up the contents of a namespace over time using a series of **namespace eval** commands. For example, the following series of commands has the same effect as the namespace definition shown above:

```
namespace eval Counter {
    variable num 0

    proc Bump {} {
        variable num
        return [incr num]
    }
}

namespace eval Counter {
    proc test {args} {
        return $args
    }
}

namespace eval Counter {
    rename test ""
}
```

Note that the **test** procedure is added to the **Counter** namespace, and later removed via the **rename** command.

Namespaces can have other namespaces within them, so they nest hierarchically. A nested namespace is encapsulated inside its parent namespace and can not interfere with other namespaces.

### Qualified names

Since namespaces may nest, qualified names are used to refer to commands, variables, and child namespaces contained inside namespaces. Qualified names are similar to the hierarchical path names for Unix files or Tk widgets, except that **::** is used as the separator instead of **/** or **.**. The topmost or global namespace has the name **``** (i.e., an empty string), although **::** is a synonym. As an example, the name **::safe::interp::create** refers to the command **create** in the namespace **interp** that is a child of of namespace **::safe**, which in turn is a child of the global namespace **::**.

If you want to access commands and variables from another namespace, you must use some extra syntax. Names must be qualified by the namespace that contains them.

From the global namespace, we might access the **Counter** procedures like this:

```
Counter::Bump 5
Counter::Reset
```

We could access the current count like this:

```
puts "count = $Counter::num"
```

When one namespace contains another, you may need more than one qualifier to reach its elements. If we had a namespace **Foo** that contained the namespace **Counter**, you could invoke its **Bump** procedure from the global namespace like this:

```
Foo::Counter::Bump 3
```

You can also use qualified names when you create and rename commands. For example, you could add a procedure to the **Foo** namespace like this:

```
proc Foo::Test {args} {return $args}
```

And you could move the same procedure to another namespace like this:

```
rename Foo::Test Bar::Test
```

There are a few remaining points about qualified names that we should cover. Namespaces have nonempty names except for the global namespace. **::** is disallowed in simple command, variable, and namespace names except as a namespace separator. Extra **:s** in a qualified name are ignored; that is, two or more **:s** are treated as a namespace separator. A trailing **::** in a qualified variable or command name refers to the variable or command named **{}**. However, a trailing **::** in a qualified namespace name is ignored.

### Name resolution

In general, all Tcl commands that take variable and command names support qualified names. This means you can give qualified names to such commands as **set**, **proc** and **rename**. If you provide a fully-qualified name that starts with a **::**, there is no question about what command, variable, or namespace you mean. However, if the name does not start with a **::** (i.e., is *relative*), Tcl follows a fixed rule for looking it up: Command and variable names are always resolved by looking first in the current namespace, and then in the global namespace. Namespace names, on the other hand, are always resolved by looking in only the current namespace.

In the following example,

```
set traceLevel 0
namespace eval Debug {
    printTrace $traceLevel
}
```

Tcl looks for **traceLevel** in the namespace **Debug** and then in the global namespace. It looks up the command **printTrace** in the same way. If a variable or command name is not found in either context, the name is undefined. To make this point absolutely clear, consider the following example:

```
set traceLevel 0
namespace eval Foo {
    variable traceLevel 3

    namespace eval Debug {
        printTrace $traceLevel
    }
}
```

Here Tcl looks for **traceLevel** first in the namespace **Foo::Debug**. Since it is not found there, Tcl then looks for it in the global namespace. The variable **Foo::traceLevel** is completely ignored during the name resolution process.

You can use the **namespace which** command to clear up any question about name resolution. For example, the command:

```
namespace eval Foo::Debug {namespace which -variable traceLevel}
```

returns **::traceLevel**. On the other hand, the command,

```
namespace eval Foo {namespace which -variable traceLevel}
```

returns `::Foo::traceLevel`.

As mentioned above, namespace names are looked up differently than the names of variables and commands. Namespace names are always resolved in the current namespace. This means, for example, that a **namespace eval** command that creates a new namespace always creates a child of the current namespace unless the new namespace name begins with a `::`.

Tcl has no access control to limit what variables, commands, or namespaces you can reference. If you provide a qualified name that resolves to an element by the name resolution rule above, you can access the element.

You can access a namespace variable from a procedure in the same namespace by using the **variable** command. Much like the **global** command, this creates a local link to the namespace variable. If necessary, it also creates the variable in the current namespace and initializes it. Note that the **global** command only creates links to variables in the global namespace. It is not necessary to use a **variable** command if you always refer to the namespace variable using an appropriate qualified name.

### **Importing commands**

Namespaces are often used to represent libraries. Some library commands are used so frequently that it is a nuisance to type their qualified names. For example, suppose that all of the commands in a package like BLT are contained in a namespace called **Blt**. Then you might access these commands like this:

```
Blt::graph .g -background red
Blt::table . .g 0,0
```

If you use the **graph** and **table** commands frequently, you may want to access them without the **Blt::** prefix. You can do this by importing the commands into the current namespace, like this:

```
namespace import Blt::*
```

This adds all exported commands from the **Blt** namespace into the current namespace context, so you can write code like this:

```
graph .g -background red
table . .g 0,0
```

The **namespace import** command only imports commands from a namespace that that namespace exported with a **namespace export** command.

Importing *every* command from a namespace is generally a bad idea since you don't know what you will get. It is better to import just the specific commands you need. For example, the command

```
namespace import Blt::graph Blt::table
```

imports only the **graph** and **table** commands into the current context.

If you try to import a command that already exists, you will get an error. This prevents you from importing the same command from two different packages. But from time to time (perhaps when debugging), you may want to get around this restriction. You may want to reissue the **namespace import** command to pick up new commands that have appeared in a namespace. In that case, you can use the **-force** option, and existing commands will be silently overwritten:

```
namespace import -force Blt::graph Blt::table
```

If for some reason, you want to stop using the imported commands, you can remove them with an **namespace forget** command, like this:

```
namespace forget Blt::*
```

This searches the current namespace for any commands imported from **Blt**. If it finds any, it removes them. Otherwise, it does nothing. After this, the **Blt** commands must be accessed with the **Blt::** prefix.

When you delete a command from the exporting namespace like this:

```
rename Blt::graph ""
```

the command is automatically removed from all namespaces that import it.

### Exporting commands

You can export commands from a namespace like this:

```
namespace eval Counter {

    namespace export Bump Reset
    variable num 0
    variable max 100

    proc Bump {{by 1}} {
        variable num
        incr num $by
        check
        return $num
    }
    proc Reset {} {
        variable num
        set num 0
    }
    proc check {} {
        variable num
        variable max
        if {$num > $max} {
            error "too high!"
        }
    }
}
```

The procedures **Bump** and **Reset** are exported, so they are included when you import from the **Counter** namespace, like this:

```
namespace import Counter::*
```

However, the **check** procedure is not exported, so it is ignored by the import operation.

The **namespace import** command only imports commands that were declared as exported by their namespace. The **namespace export** command specifies what commands may be imported by other namespaces. If a **namespace import** command specifies a command that is not exported, the command is not imported.

### See also

**variable**

## 2.44 open - Open a File-Based or Command Pipeline Channel

### Name

**open** - Open a file-based or command pipeline channel

### Synopsis

**open** *fileName*

**open** *fileName access*

**open** *fileName access permissions*

### Description

This command opens a file, serial port, or command pipeline and returns a channel identifier that may be used in future invocations of commands like **read**, **puts**, and **close**. If the first character of *fileName* is not | then the command opens a file: *fileName* gives the name of the file to open.

The *access* argument, if present, indicates the way in which the file (or command pipeline) is to be accessed. In the first form *access* may have any of the following values:

**r** ..... Open the file for reading only; the file must already exist. This is the default value if *access* is not specified.

**r+** ..... Open the file for both reading and writing; the file must already exist.

**w** ..... Open the file for writing only. Truncate it if it exists. If it doesn't exist, create a new file.

**w+** ..... Open the file for reading and writing. Truncate it if it exists. If it doesn't exist, create a new file.

**a** ..... Open the file for writing only. The file must already exist, and the file is positioned so that new data is appended to the file.

**a+** ..... Open the file for reading and writing. If the file doesn't exist, create a new empty file. Set the initial access position to the end of the file.

In the second form, *access* consists of a list of any of the following flags, all of which have the standard POSIX meanings. One of the flags must be either **RDONLY**, **WRONLY** or **RDWR**.

**RDONLY** ..... Open the file for reading only.

**WRONLY** ..... Open the file for writing only.

**RDWR** ..... Open the file for both reading and writing.

**APPEND** ..... Set the file pointer to the end of the file prior to each write.

**CREAT** ..... Create the file if it doesn't already exist (without this flag it is an error for the file not to exist).

**EXCL** ..... If **CREAT** is also specified, an error is returned if the file already exists.

**NOCTTY** ..... If the file is a terminal device, this flag prevents the file from becoming the controlling terminal of the process.

**NONBLOCK** ..... Prevents the process from blocking while opening the file, and possibly in subsequent I/O operations. The exact behavior of this flag is system- and device-dependent; its use is discouraged (it is better to use the **fconfigure** command to put a file in nonblocking

mode). For details refer to your system documentation on the **open** system call's **O\_NONBLOCK** flag.

**TRUNC** .....If the file exists it is truncated to zero length.

If a new file is created as part of opening it, *permissions* (an integer) is used to set the permissions for the new file in conjunction with the process's file mode creation mask. *Permissions* defaults to 0666.

Example to write in a file:

```
set f [open $Path "w"]
puts $f "Hello word. Welcome ! "
puts $f "456"
puts $f "Last line of file."
close $f
```

### Command pipelines

If the first character of *fileName* is ``|` then the remaining characters of *fileName* are treated as a list of arguments that describe a command pipeline to invoke, in the same style as the arguments for **exec**. In this case, the channel identifier returned by **open** may be used to write to the command's input pipe or read from its output pipe, depending on the value of *access*. If write-only access is used (e.g. *access* is **w**), then standard output for the pipeline is directed to the current standard output unless overridden by the command. If read-only access is used (e.g. *access* is **r**), standard input for the pipeline is taken from the current standard input unless overridden by the command.

### Serial communications

If *fileName* refers to a serial port, then the specified serial port is opened and initialized in a platform-dependent manner. Acceptable values for the *fileName* to use to open a serial port are described in the PORTABILITY ISSUES section.

### Configuration options

The **fconfigure** command can be used to query and set the following configuration option for open serial ports:

**-mode** baud,parity,data,stop

This option is a set of 4 comma-separated values: the baud rate, parity, number of data bits, and number of stop bits for this serial port. The baud rate is a simple integer that specifies the connection speed. Parity is one of the following letters: **n, o, e, m, s**; respectively signifying the parity options of ``none'`, ``odd'`, ``even'`, ``mark'`, or ``space'`. Data is the number of data bits and should be an integer from 5 to 8, while stop is the number of stop bits and should be the integer 1 or 2.

### Portability issues

**Unix:** Valid values for *fileName* to open a serial port are generally of the form `/dev/ttyX`, where *X* is **a** or **b**, but the name of any pseudo-file that maps to a serial port may be used.

When running Tcl interactively, there may be some strange interactions between the console, if one is present, and a command pipeline that uses standard input. If a command pipeline is opened for reading, some of the lines entered at the console will be sent to the command pipeline and some will be sent to the Tcl evaluator. This problem only occurs because both Tcl and the child application are competing for the console at the same time. If the command pipeline is started from a script, so that Tcl is not accessing the console, or if the command pipeline does not use standard input, but is redirected from a file, then the above problem does not occur.

See the PORTABILITY ISSUES section of the `exec` command for additional information not specific to command pipelines about executing applications on the various platforms

**See also**

`close`, `filename`, `gets`, `read`, `puts`, `exec`

## 2.45 **pid - Retrieve Process id(s)**

### Name

**pid** - Retrieve process id(s)

### Synopsis

**pid** ?*fileId*?

### Description

If the *fileId* argument is given then it should normally refer to a process pipeline created with the **open** command. In this case the **pid** command will return a list whose elements are the process identifiers of all the processes in the pipeline, in order. The list will be empty if *fileId* refers to an open file that isn't a process pipeline. If no *fileId* argument is given then **pid** returns the process identifier of the current process. All process identifiers are returned as decimal strings.

## 2.46 **proc** - Create a Tcl Procedure

### Name

**proc** - Create a Tcl procedure

### Synopsis

**proc** *name args body*

### Description

The **proc** command creates a new Tcl procedure named *name*, replacing any existing command or procedure there may have been by that name. Whenever the new command is invoked, the contents of *body* will be executed by the Tcl interpreter. Normally, *name* is unqualified (does not include the names of any containing namespaces), and the new procedure is created in the current namespace. If *name* includes any namespace qualifiers, the procedure is created in the specified namespace. *Args* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value.

When *name* is invoked a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name **args**, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to **args** are combined into a list (as if the **list** command had been used); this combined value is assigned to the local variable **args**.

When *body* is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the **global** command or the **upvar** command. Namespace variables can only be accessed by invoking the **variable** command or the **upvar** command.

The **proc** command returns an empty string. When a procedure is invoked, the procedure's return value is the value specified in a **return** command. If the procedure doesn't execute an explicit **return**, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure-as-a-whole will return that same error.

## 2.47 puts - Write to a Channel

### Name

**puts** - Write to a channel

### Synopsis

**puts** *?-nonewline? ?channelId? string*

### Description

Writes the characters given by *string* to the channel given by *channelId*. *ChannelId* must be a channel identifier such as returned from a previous invocation of **open** or **socket**. It must have been opened for output. If no *channelId* is specified then it defaults to **stdout**. **Puts** normally outputs a newline character after *string*, but this feature may be suppressed by specifying the **-nonewline** switch.

Newline characters in the output are translated by **puts** to platform-specific end-of-line sequences according to the current value of the **-translation** option for the channel (for example, on PCs newlines are normally replaced with carriage-return-linefeed sequences). See the **configure** manual entry for a discussion of end-of-line translations.

Tcl buffers output internally, so characters written with **puts** may not appear immediately on the output file or device; Tcl will normally delay output until the buffer is full or the channel is closed. You can force output to appear immediately with the **flush** command.

When the output buffer fills up, the **puts** command will normally block until all the buffered data has been accepted for output by the operating system. If *channelId* is in nonblocking mode then the **puts** command will not block even if the operating system cannot accept the data. Instead, Tcl continues to buffer the data and writes it in the background as fast as the underlying file or device can accept it. The application must use the Tcl event loop for nonblocking output to work; otherwise Tcl never finds out that the file or device is ready for more output data. It is possible for an arbitrarily large amount of data to be buffered for a channel in nonblocking mode, which could consume a large amount of memory. To avoid wasting memory, nonblocking I/O should normally be used in an event-driven fashion with the **fileevent** command (don't invoke **puts** unless you have recently been notified via a file event that the channel is ready for more output data).

### See also

**fileevent**

## 2.48 **pwd - Return the Current Working Directory**

### Name

**pwd** - Return the current working directory

### Synopsis

**pwd**

### Description

Returns the path name of the current working directory.

Example:

```
set currentPath [ pwd ]  
puts "Current path is $currentPath"
```

## 2.49 read - Read from a Channel

### Name

**read** - Read from a channel

### Synopsis

**read** *?-nonewline?* *channelId*

**read** *channelId* *numChars*

### Description

In the first form, the **read** command reads all of the data from *channelId* up to the end of the file. If the **-nonewline** switch is specified then the last character of the file is discarded if it is a newline. In the second form, the extra argument specifies how many characters to read. Exactly that many characters will be read and returned, unless there are fewer than *numChars* left in the file; in this case all the remaining characters are returned. If the channel is configured to use a multi-byte encoding, then the number of characters read may not be the same as the number of bytes read.

*ChannelId* must be an identifier for an open channel such as the Tcl standard input channel (**stdin**), the return value from an invocation of **open** or **socket**, or the result of a channel creation command provided by a Tcl extension. The channel must have been opened for input.

If *channelId* is in nonblocking mode, the command may not read as many characters as requested: once all available input has been read, the command will return the data that is available rather than blocking for more input. If the channel is configured to use a multi-byte encoding, then there may actually be some bytes remaining in the internal buffers that do not form a complete character. These bytes will not be returned until a complete character is available or end-of-file is reached. The **-nonewline** switch is ignored if the command returns before reaching the end of the file.

**Read** translates end-of-line sequences in the input into newline characters according to the **-translation** option for the channel. See the **fconfigure** manual entry for a discussion on ways in which **fconfigure** will alter input.

### USE WITH SERIAL PORTS

For most applications a channel connected to a serial port should be configured to be nonblocking: **fconfigure** *channelId* **-blocking 0**. Then **read** behaves much like described above. Care must be taken when using **read** on blocking serial ports:

**read** *channelId* *numChars*

In this form **read** blocks until *numChars* have been received from the serial port.

**read** *channelId*

In this form **read** blocks until the reception of the end-of-file character, see **fconfigure -eofchar**. If there no end-of-file character has been configured for the channel, then **read** will block forever.

### See also

**file**, **eof**, **fconfigure**

## 2.50 **rename - Rename or Delete a Command**

### **Name**

**rename** - Rename or delete a command

### **Synopsis**

**rename** *oldName newName*

### **Description**

Rename the command that used to be called *oldName* so that it is now called *newName*. If *newName* is an empty string then *oldName* is deleted. *oldName* and *newName* may include namespace qualifiers (names of containing namespaces). If a command is renamed into a different namespace, future invocations of it will execute in the new namespace. The **rename** command returns an empty string as result.

### **See also**

**namespace, proc**

## 2.51 return - Return from a Procedure

### Name

**return** - Return from a procedure

### Synopsis

**return** *?-code code? ?-errorinfo info? ?-errorcode code? ?string?*

### Description

Return immediately from the current procedure (or top-level command or **source** command), with *string* as the return value. If *string* is not specified then an empty string will be returned as result.

### Exceptional returns

In the usual case where the **-code** option isn't specified the procedure will return normally (its completion code will be `TCL_OK`). However, the **-code** option may be used to generate an exceptional return from the procedure. *Code* may have any of the following values:

#### **ok**

Normal return: same as if the option is omitted.

#### **error**

Error return: same as if the **error** command were used to terminate the procedure, except for handling of **errorInfo** and **errorCode** variables (see below).

#### **return**

The current procedure will return with a completion code of `TCL_RETURN`, so that the procedure that invoked it will return also.

#### **break**

The current procedure will return with a completion code of `TCL_BREAK`, which will terminate the innermost nested loop in the code that invoked the current procedure.

#### **continue**

The current procedure will return with a completion code of `TCL_CONTINUE`, which will terminate the current iteration of the innermost nested loop in the code that invoked the current procedure.

#### *value*

*Value* must be an integer; it will be returned as the completion code for the current procedure.

The **-code** option is rarely used. It is provided so that procedures that implement new control structures can reflect exceptional conditions back to their callers.

Two additional options, **-errorinfo** and **-errorcode**, may be used to provide additional information during error returns. These options are ignored unless *code* is **error**.

The **-errorinfo** option specifies an initial stack trace for the **errorInfo** variable; if it is not specified then the stack trace left in **errorInfo** will include the call to the procedure and higher levels on the stack but it will not include any information about the context of the error within the procedure. Typically the *info* value is supplied from the value left in **errorInfo** after a **catch** command trapped an error within the procedure.

If the **-errorcode** option is specified then *code* provides a value for the **errorCode** variable. If the option is not specified then **errorCode** will default to **NONE**.

**See also**

**break, continue, error, proc**

## 2.52 **scan - Parse String Using Conversion Specifiers in the Style of sscanf**

### Name

**scan** - Parse string using conversion specifiers in the style of sscanf

### Synopsis

**scan** *string format ?varName varName ...?*

### Introduction

This command parses fields from an input string in the same fashion as the ANSI C **sscanf** procedure and returns a count of the number of conversions performed, or -1 if the end of the input string is reached before any conversions have been performed. *String* gives the input to be parsed and *format* indicates how to parse it, using % conversion specifiers as in **sscanf**. Each *varName* gives the name of a variable; when a field is scanned from *string* the result is converted back into a string and assigned to the corresponding variable. If no *varName* variables are specified, then **scan** works in an inline manner, returning the data that would otherwise be stored in the variables as a list. In the inline case, an empty string is returned when the end of the input string is reached before any conversions have been performed.

### Details on scanning

**Scan** operates by scanning *string* and *format* together. If the next character in *format* is a blank or tab then it matches any number of white space characters in *string* (including zero). Otherwise, if it isn't a % character then it must match the next character of *string*. When a % is encountered in *format*, it indicates the start of a conversion specifier. A conversion specifier contains up to four fields after the %: a \*, which indicates that the converted value is to be discarded instead of assigned to a variable; a XPG3 position specifier; a number indicating a maximum field width; a field size modifier; and a conversion character. All of these fields are optional except for the conversion character. The fields that are present must appear in the order given above. When **scan** finds a conversion specifier in *format*, it first skips any white-space characters in *string* (unless the specifier is | or c). Then it converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to **scan**.

If the % is followed by a decimal number and a \$, as in ``%2\$d'', then the variable to use is not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where 1 corresponds to the first *varName*. If there are any positional specifiers in *format* then all of the specifiers must be positional. Every *varName* on the argument list must correspond to exactly one conversion specifier or an error is generated, or in the inline case, any position can be specified at most once and the empty positions will be filled in with empty strings.

The following conversion characters are supported:

#### **d**

The input field must be a decimal integer. It is read in and the value is stored in the variable as a decimal string. If the l or L field size modifier is given, the scanned value will have an internal representation that is at least 64-bits in size.

#### **o**

The input field must be an octal integer. It is read in and the value is stored in the variable as a decimal string. If the l or L field size modifier is given, the scanned value will have an internal representation that is at least 64-bits in size. If the value exceeds MAX\_INT (017777777777 on platforms using 32-bit integers when the l and L

modifiers are not given), it will be truncated to a signed integer. Hence, 037777777777 will appear as -1 on a 32-bit machine by default.

**x**

The input field must be a hexadecimal integer. It is read in and the value is stored in the variable as a decimal string. If the **I** or **L** field size modifier is given, the scanned value will have an internal representation that is at least 64-bits in size. If the value exceeds `MAX_INT` (0x7FFFFFFF on platforms using 32-bit integers when the **I** and **L** modifiers are not given), it will be truncated to a signed integer. Hence, 0xFFFFFFFF will appear as -1 on a 32-bit machine.

**u**

The input field must be a decimal integer. The value is stored in the variable as an unsigned decimal integer string. If the **I** or **L** field size modifier is given, the scanned value will have an internal representation that is at least 64-bits in size.

**i**

The input field must be an integer. The base (i.e. decimal, octal, or hexadecimal) is determined in the same fashion as described in **expr**. The value is stored in the variable as a decimal string. If the **I** or **L** field size modifier is given, the scanned value will have an internal representation that is at least 64-bits in size.

**c**

A single character is read in and its binary value is stored in the variable as a decimal string. Initial white space is not skipped in this case, so the input field may be a white-space character. This conversion is different from the ANSI standard in that the input field always consists of a single character and no field width may be specified.

**s**

The input field consists of all the characters up to the next white-space character; the characters are copied to the variable.

**e or f or g**

The input field must be a floating-point number consisting of an optional sign, a string of decimal digits possibly containing a decimal point, and an optional exponent consisting of an **e** or **E** followed by an optional sign and a string of decimal digits. It is read in and stored in the variable as a floating-point string.

**[chars]**

The input field consists of any number of characters in *chars*. The matching string is stored in the variable. If the first character between the brackets is a **]** then it is treated as part of *chars* rather than the closing bracket for the set. If *chars* contains a sequence of the form *a-b* then any character between *a* and *b* (inclusive) will match. If the first or last character between the brackets is a **-**, then it is treated as part of *chars* rather than indicating a range.

**[^chars]**

The input field consists of any number of characters not in *chars*. The matching string is stored in the variable. If the character immediately following the **^** is a **]** then it is treated as part of the set rather than the closing bracket for the set. If *chars* contains a sequence of the form *a-b* then any character between *a* and *b* (inclusive) will be excluded from the set. If the first or last character between the brackets is a **-**, then it is treated as part of *chars* rather than indicating a range.

**n**

No input is consumed from the input string. Instead, the total number of characters scanned from the input string so far is stored in the variable.

The number of characters read from the input for a conversion is the largest number that makes sense for that particular conversion (e.g. as many decimal

digits as possible for **%d**, as many octal digits as possible for **%o**, and so on). The input field for a given conversion terminates either when a white-space character is encountered or when the maximum field width has been reached, whichever comes first. If a **\*** is present in the conversion specifier then no variable is assigned and the next scan argument is not consumed.

### **Differences from ansi sscanf**

The behavior of the **scan** command is the same as the behavior of the ANSI C **sscanf** procedure except for the following differences:

[1]

**%p** conversion specifier is not currently supported.

[2]

For **%c** conversions a single character value is converted to a decimal string, which is then assigned to the corresponding *varName*; no field width may be specified for this conversion.

[3]

The **h** modifier is always ignored and the **l** and **L** modifiers are ignored when converting real values (i.e. type **double** is used for the internal representation).

[4]

If the end of the input string is reached before any conversions have been performed and no variables are given, an empty string is returned.

### **See also**

**format**, **sscanf**

## 2.53 seek - Change the Access Position for an Open Channel

### Name

**seek** - Change the access position for an open channel

### Synopsis

**seek** *channelId* *offset* *?origin?*

### Description

Changes the current access position for *channelId*.

*ChannelId* must be an identifier for an open channel such as a Tcl standard channel (**stdin**, **stdout**, or **stderr**), the return value from an invocation of **open** or **socket**, or the result of a channel creation command provided by a Tcl extension.

The *offset* and *origin* arguments specify the position at which the next read or write will occur for *channelId*. *Offset* must be an integer (which may be negative) and *origin* must be one of the following:

#### **start**

The new access position will be *offset* bytes from the start of the underlying file or device.

#### **current**

The new access position will be *offset* bytes from the current access position; a negative *offset* moves the access position backwards in the underlying file or device.

#### **end**

The new access position will be *offset* bytes from the end of the file or device. A negative *offset* places the access position before the end of file, and a positive *offset* places the access position after the end of file.

The *origin* argument defaults to **start**.

The command flushes all buffered output for the channel before the command returns, even if the channel is in nonblocking mode. It also discards any buffered and unread input. This command returns an empty string. An error occurs if this command is applied to channels whose underlying file or device does not support seeking.

Note that *offset* values are byte offsets, not character offsets. Both **seek** and **tell** operate in terms of bytes, not characters, unlike **read**.

### See also

**file**, **open**, **close**, **gets**, **tell**

## 2.54 **set - Read and Write Variables**

### **Name**

**set** - Read and write variables

### **Synopsis**

**set** *varName* ?*value*?

### **Description**

Returns the value of variable *varName*. If *value* is specified, then set the value of *varName* to *value*, creating a new variable if one doesn't already exist, and return its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the **global** command was invoked to declare *varName* to be global, or unless a **variable** command was invoked to declare *varName* to be a namespace variable.

### **See also**

**expr, proc, trace, unset**

## 2.55 **socket - Open a TCP Network Connection**

### Name

**socket** - Open a TCP network connection

### Synopsis

**socket** *?options? host port*

**socket -server** *command ?options? port*

### Description

This command opens a network socket and returns a channel identifier that may be used in future invocations of commands like **read**, **puts** and **flush**. At present only the TCP network protocol is supported; future releases may include support for additional protocols. The **socket** command may be used to open either the client or server side of a connection, depending on whether the **-server** switch is specified.

Note that the default encoding for *all* sockets is the system encoding, as returned by **encoding system**. Most of the time, you will need to use **fconfigure** to alter this to something else, such as *utf-8* (ideal for communicating with other Tcl processes) or *iso8859-1* (useful for many network protocols, especially the older ones).

### Client sockets

If the **-server** option is not specified, then the client side of a connection is opened and the command returns a channel identifier that can be used for both reading and writing. *Port* and *host* specify a port to connect to; there must be a server accepting connections on this port. *Port* is an integer port number (or service name, where supported and understood by the host operating system) and *host* is either a domain-style name such as **www.sunlabs.com** or a numerical IP address such as **127.0.0.1**. Use *localhost* to refer to the host on which the command is invoked.

The following options may also be present before *host* to specify additional information about the connection:

#### **-myaddr** *addr*

*Addr* gives the domain-style name or numerical IP address of the client-side network interface to use for the connection. This option may be useful if the client machine has multiple network interfaces. If the option is omitted then the client-side interface will be chosen by the system software.

#### **-myport** *port*

*Port* specifies an integer port number (or service name, where supported and understood by the host operating system) to use for the client's side of the connection. If this option is omitted, the client's port number will be chosen at random by the system software.

#### **-async**

The **-async** option will cause the client socket to be connected asynchronously. This means that the socket will be created immediately but may not yet be connected to the server, when the call to **socket** returns. When a **gets** or **flush** is done on the socket before the connection attempt succeeds or fails, if the socket is in blocking mode, the operation will wait until the connection is completed or fails. If the socket is in nonblocking mode and a **gets** or **flush** is done on the socket before the connection attempt succeeds or fails.

### Server sockets

If the **-server** option is specified then the new socket will be a server for the port given by *port* (either an integer or a service name, where supported and understood by the host operating system). Tcl will automatically accept connections to the given port. For each connection Tcl will create a new channel that may be used to communicate with the client. Tcl then invokes *command* with three additional arguments: the name of the new channel, the address, in network address notation, of the client's host, and the client's port number.

The following additional option may also be specified before *host*:

#### **-myaddr** *addr*

*Addr* gives the domain-style name or numerical IP address of the server-side network interface to use for the connection. This option may be useful if the server machine has multiple network interfaces. If the option is omitted then the server socket is bound to the special address INADDR\_ANY so that it can accept connections from any interface.

Server channels cannot be used for input or output; their sole use is to accept new client connections. The channels created for each incoming client connection are opened for input and output. Closing the server channel shuts down the server so that no new connections will be accepted; however, existing connections will be unaffected.

Server sockets depend on the Tcl event mechanism to find out when new connections are opened. If the application doesn't enter the event loop, for example by invoking the **vwait** command, then no connections will be accepted.

If *port* is specified as zero, the operating system will allocate an unused port for use as a server socket. The port number actually allocated may be retrieved from the created server socket using the **fconfigure** command to retrieve the **-sockname** option as described below.

### Configuration options

The **fconfigure** command can be used to query several readonly configuration options for socket channels:

#### **-error**

This option gets the current error status of the given socket. This is useful when you need to determine if an asynchronous connect operation succeeded. If there was an error, the error message is returned. If there was no error, an empty string is returned.

#### **-sockname**

This option returns a list of three elements, the address, the host name and the port number for the socket. If the host name cannot be computed, the second element is identical to the address, the first element of the list.

#### **-peername**

This option is not supported by server sockets. For client and accepted sockets, this option returns a list of three elements; these are the address, the host name and the port to which the peer socket is connected or bound. If the host name cannot be computed, the second element of the list is identical to the address, its first element.

### See also

**fconfigure**, **flush**, **open**, **read**

## 2.56 **source - Evaluate a File or Resource as a Tcl Script**

### **Name**

**source** - Evaluate a file or resource as a Tcl script

### **Synopsis**

**source** *fileName*

### **Description**

This command takes the contents of the specified file or resource and passes it to the Tcl interpreter as a text script. The return value from **source** is the return value of the last command executed in the script. If an error occurs in evaluating the contents of the script then the **source** command will return that error. If a **return** command is invoked from within the script then the remainder of the file will be skipped and the **source** command will return normally with the result from the **return** command.

The end-of-file character for files is '\32' (^Z) for all platforms. The source command will read files up to this character. This restriction does not exist for the **read** or **gets** commands, allowing for files containing code and data segments (scripted documents). If you require a ``^Z" in code for string comparison, you can use ``\032" or ``\u001a", which will be safely substituted by the Tcl interpreter into ``^Z".

### **See also**

**file, cd**

## 2.57 **split** - Split a STRING into a PROPER Tcl LIST

### Name

**split** - Split a string into a proper Tcl list

### Synopsis

**split** *string* ?*splitChars*?

### Description

Returns a list created by splitting *string* at each character that is in the *splitChars* argument. Each element of the result list will consist of the characters from *string* that lie between instances of the characters in *splitChars*. Empty list elements will be generated if *string* contains adjacent characters in *splitChars*, or if the first or last character of *string* is in *splitChars*. If *splitChars* is an empty string then each character of *string* becomes a separate element of the result list. *SplitChars* defaults to the standard white-space characters.

For example,

```
split "comp.unix.misc" .
```

returns "**comp unix misc**" and

```
split "Hello world" {}
```

returns "**H e l l o { } w o r l d**".

### See also

**join**, **list**, **string**

## 2.58 **string** - Manipulate Strings

### Name

**string** - Manipulate strings

### Synopsis

**string** *option arg ?arg ...?*

### Description

Performs one of several string operations, depending on *option*.

The legal *options* (which may be abbreviated) are:

**string bytlength** *string*

Returns a decimal string giving the number of bytes used to represent *string* in memory. Because UTF-8 uses one to three bytes to represent Unicode characters, the byte length will not be the same as the character length in general. The cases where a script cares about the byte length are rare. In almost all cases, you should use the **string length** operation (including determining the length of a Tcl ByteArray object).

**string compare** *?-nocase? ?-length int? string1 string2*

Perform a character-by-character comparison of strings *string1* and *string2*. Returns -1, 0, or 1, depending on whether *string1* is lexicographically less than, equal to, or greater than *string2*. If **-length** is specified, then only the first *length* characters are used in the comparison. If **-length** is negative, it is ignored. If **-nocase** is specified, then the strings are compared in a case-insensitive manner.

**string equal** *?-nocase? ?-length int? string1 string2*

Perform a character-by-character comparison of strings *string1* and *string2*. Returns 1 if *string1* and *string2* are identical, or 0 when not. If **-length** is specified, then only the first *length* characters are used in the comparison. If **-length** is negative, it is ignored. If **-nocase** is specified, then the strings are compared in a case-insensitive manner.

**string first** *string1 string2 ?startIndex?*

Search *string2* for a sequence of characters that exactly match the characters in *string1*. If found, return the index of the first character in the first such match within *string2*. If not found, return -1. If *startIndex* is specified (in any of the forms accepted by the **index** method), then the search is constrained to start with the character in *string2* specified by the index.

For example,

```
string first a 0a23456789abcdef 5
```

will return **10**, but

```
string first a 0123456789abcdef 11
```

will return **-1**.

**string index** *string charIndex*

Returns the *charIndex*'th character of the *string* argument.

A *charIndex* of 0 corresponds to the first character of the string. *charIndex* may be specified as follows:

*integer*

The char specified at this integral index.

**end**

The last char of the string.

**end-integer**

The last char of the string minus the specified integer offset (e.g. **end-1** would refer to the "c" in "abcd").

If *charIndex* is less than 0 or greater than or equal to the length of the string then an empty string is returned.

**string is class ?-strict? ?-failindex varname? string**

Returns 1 if *string* is a valid member of the specified character class, otherwise returns 0.

If **-strict** is specified, then an empty string returns 0, otherwise an empty string will return 1 on any class.

If **-failindex** is specified, then if the function returns 0, the index in the string where the class was no longer valid will be stored in the variable named *varname*. The *varname* will not be set if the function returns 1. The following character classes are recognized (the class name can be abbreviated):

**alnum**

Any Unicode alphabet or digit character.

**alpha**

Any Unicode alphabet character.

**ascii**

Any character with a value less than \u0080 (those that are in the 7-bit ascii range).

**boolean**

Any of the forms allowed to **Tcl\_GetBoolean**.

**control**

Any Unicode control character.

**digit**

Any Unicode digit character. Note that this includes characters outside of the [0-9] range.

**double**

Any of the valid forms for a double in Tcl, with optional surrounding whitespace. In case of under/overflow in the value, 0 is returned and the *varname* will contain -1.

**false**

Any of the forms allowed to **Tcl\_GetBoolean** where the value is false.

**graph**

Any Unicode printing character, except space.

**integer**

Any of the valid forms for a 32-bit integer in Tcl, with optional surrounding whitespace. In case of under/overflow in the value, 0 is returned and the *varname* will contain -1.

**lower**

Any Unicode lower case alphabet character.

**print**

Any Unicode printing character, including space.

**punct**

Any Unicode punctuation character.

**space**

Any Unicode space character.

**true**

Any of the forms allowed to **Tcl\_GetBoolean** where the value is true.

**upper**

Any upper case alphabet character in the Unicode character set.

**wordchar**

Any Unicode word character. That is any alphanumeric character, and any Unicode connector punctuation characters (e.g. underscore).

**xdigit**

Any hexadecimal digit character ([0-9A-Fa-f]).

In the case of **boolean**, **true** and **false**, if the function will return 0, then the *varname* will always be set to 0, due to the varied nature of a valid boolean value.

**string last** *string1 string2 ?lastIndex?*

Search *string2* for a sequence of characters that exactly match the characters in *string1*. If found, return the index of the first character in the last such match within *string2*. If there is no match, then return -1. If *lastIndex* is specified (in any of the forms accepted by the **index** method), then only the characters in *string2* at or before the specified *lastIndex* will be considered by the search.

For example,

```
string last a 0a23456789abcdef 15
```

will return **10**, but

```
string last a 0a23456789abcdef 9
```

will return **1**.

**string length** *string*

Returns a decimal string giving the number of characters in *string*. Note that this is not necessarily the same as the number of bytes used to store the string. If the object is a ByteArray object (such as those returned from reading a binary encoded channel), then this will return the actual byte length of the object.

**string map** *?-nocase? charMap string*

Replaces characters in *string* based on the key-value pairs in *charMap*. *charMap* is a list of *key value key value ...* as in the form returned by **array get**. Each instance of a key in the string will be replaced with its corresponding value. If **-nocase** is specified, then matching is done without regard to case differences. Both *key* and *value* may be multiple characters. Replacement is done in an ordered manner, so the key appearing first in the list will be checked first, and so on. *string* is only iterated over once, so earlier key replacements will have no affect for later key matches.

For example,

```
string map {abc 1 ab 2 a 3 1 0} 1abcaababcabababc
```

will return the string **01321221**.

**string match** *?-nocase? pattern string*

See if *pattern* matches *string*; return 1 if it does, 0 if it doesn't.

If **-nocase** is specified, then the pattern attempts to match against the string in a case insensitive manner.

For the two strings to match, their contents must be identical except that the following special sequences may appear in *pattern*:

\*

Matches any sequence of characters in *string*, including a null string.

?

Matches any single character in *string*.

[*chars*]

Matches any character in the set given by *chars*. If a sequence of the form *x-y* appears in *chars*, then any character between *x* and *y*, inclusive, will match. When used with **-nocase**, the end points of the range are converted to lower case first. Whereas {[A-z]} matches '\_' when matching case-sensitively ('\_' falls between the 'Z' and 'a'), with **-nocase** this is considered like {[A-Za-z]} (and probably what was meant in the first place).

\x

Matches the single character *x*. This provides a way of avoiding the special interpretation of the characters \*?[\ in *pattern*.

**string range** *string first last*

Returns a range of consecutive characters from *string*, starting with the character whose index is *first* and ending with the character whose index is *last*. An index of 0 refers to the first character of the string. *first* and *last* may be specified as for the **index** method. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were **end**. If *first* is greater than *last* then an empty string is returned.

**string repeat** *string count*

Returns *string* repeated *count* number of times.

**string replace** *string first last ?newstring?*

Removes a range of consecutive characters from *string*, starting with the character whose index is *first* and ending with the character whose index is *last*. An index of 0 refers to the first character of the string. *first* and *last* may be specified as for the **index** method. If *newstring* is specified, then it is placed in the removed character range. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were **end**. If *first* is greater than *last* or the length of the initial string, or *last* is less than 0, then the initial string is returned untouched.

**string tolower** *string ?first? ?last?*

Returns a value equal to *string* except that all upper (or title) case letters have been converted to lower case. If *first* is specified, it refers to the first char index in the string to start modifying. If *last* is specified, it refers to the char index in the string to stop at (inclusive). *first* and *last* may be specified as for the **index** method.

**string totitle** *string ?first? ?last?*

Returns a value equal to *string* except that the first character in *string* is converted to its Unicode title case variant (or upper case if there is no title case variant) and the rest of the string is converted to lower case. If *first* is specified, it refers to the first char index in the string to start modifying. If *last* is specified, it refers to the char index in the string to stop at (inclusive). *first* and *last* may be specified as for the **index** method.

**string toupper** *string ?first? ?last?*

Returns a value equal to *string* except that all lower (or title) case letters have been converted to upper case. If *first* is specified, it refers to the first char index in the string to start modifying. If *last* is specified, it refers to the char index in the string to stop at (inclusive). *first* and *last* may be specified as for the **index** method.

**string trim** *string ?chars?*

Returns a value equal to *string* except that any leading or trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

**string trimleft** *string ?chars?*

Returns a value equal to *string* except that any leading characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

**string trimright** *string ?chars?*

Returns a value equal to *string* except that any trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

**string wordend** *string charIndex*

Returns the index of the character just after the last one in the word containing character *charIndex* of *string*. *charIndex* may be specified as for the **index** method. A word is considered to be any contiguous range of alphanumeric (Unicode letters or decimal digits) or underscore (Unicode connector punctuation) characters, or any single character other than these.

**string wordstart** *string charIndex*

Returns the index of the first character in the word containing character *charIndex* of *string*. *charIndex* may be specified as for the **index** method. A word is considered to be any contiguous range of alphanumeric (Unicode letters or decimal digits) or underscore (Unicode connector punctuation) characters, or any single character other than these.

General example:

```
set str "It's my string"
puts "The string is: $str"
puts "The length of string is: [string length $str]"
puts "The character 3 is: [string index $str 3]"
puts "The characters between 4 to 8 are: [string range $str 4 8]"
puts "The first occurrence of the character \"a\" is: [string first a $str]"
puts "The result is: [split $str {}]"
set myformat [ format "%d %s %f" 15 "it's my string" 5.12]
puts $myformat
```

=> *The string is: It's my string*

=> *The length of string: 30*

=> *The character 3 is : s*

=> *The characters between 4 to 8 are : t une*

=> *The first occurrence of the character "a" is : 12*

=> *The result is: I t ' s { } a { } s t r i n g*

=> *15 it's my string 5.120000*

**See also**

**expr, list**

## 2.59 **subst - Perform Backslash, Command, and Variable Substitutions**

### Name

**subst** - Perform backslash, command, and variable substitutions

### Synopsis

**subst** *?-noblackslashes? ?-nocommands? ?-novariables?* *string*

### Description

This command performs variable substitutions, command substitutions, and backslash substitutions on its *string* argument and returns the fully-substituted result. The substitutions are performed in exactly the same way as for Tcl commands. As a result, the *string* argument is actually substituted twice, once by the Tcl parser in the usual fashion for Tcl commands, and again by the *subst* command.

If any of the **-noblackslashes**, **-nocommands**, or **-novariables** are specified, then the corresponding substitutions are not performed. For example, if **-nocommands** is specified, command substitution is not performed: open and close brackets are treated as ordinary characters with no special interpretation.

Note that the substitution of one kind can include substitution of other kinds. For example, even when the **-novariables** option is specified, command substitution is performed without restriction. This means that any variable substitution necessary to complete the command substitution will still take place. Likewise, any command substitution necessary to complete a variable substitution will take place, even when **-nocommands** is specified. See the EXAMPLES below.

If an error occurs during substitution, then **subst** will return that error. If a break exception occurs during command or variable substitution, the result of the whole substitution will be the string (as substituted) up to the start of the substitution that raised the exception. If a continue exception occurs during the evaluation of a command or variable substitution, an empty string will be substituted for that entire command or variable substitution (as long as it is well-formed Tcl.) If a return exception occurs, or any other return code is returned during command or variable substitution, then the returned value is substituted for that substitution. See the EXAMPLES below. In this way, all exceptional return codes are "caught" by **subst**. The **subst** command itself will either return an error, or will complete successfully.

### Examples

When it performs its substitutions, *subst* does not give any special treatment to double quotes or curly braces (except within command substitutions) so the script

```
set a 44
subst {xyz {$a}}
```

returns `xyz {44}`, not `xyz {$a}`

and the script

```
set a "p} q {r"
subst {xyz {$a}}
```

return `xyz {p} q {r}`, not `xyz {p} q {r}`.

When command substitution is performed, it includes any variable substitution necessary to evaluate the script.

```
set a 44
subst -novariables {$a [format $a]}
```

returns ```$a 44`", not ```$a $a`".

Similarly, when variable substitution is performed, it includes any command substitution necessary to retrieve the value of the variable.

```
proc b {} {return c}
array set a {c c [b] tricky}
subst -nocommands {[b] $a([b])}
```

returns ```[b] c`", not ```[b] tricky`".

The `continue` and `break` exceptions allow command substitutions to prevent substitution of the rest of the command substitution and the rest of *string* respectively, giving script authors more options when processing text using *subst*.

For example, the script

```
subst {abc,[break],def}
```

returns ```abc,`", not ```abc,,def`"

and the script

```
subst {abc,[continue;expr 1+2],def}
```

returns ```abc,,def`", not ```abc,3,def`".

Other exceptional return codes substitute the returned value

```
subst {abc,[return foo;expr 1+2],def}
```

returns ```abc,foo,def`", not ```abc,3,def`"

and

```
subst {abc,[return -code 10 foo;expr 1+2],def}
```

also returns ```abc,foo,def`", not ```abc,3,def`".

### See also

Tcl, eval, break, continue

## 2.60 **switch** - Evaluate One of Several Scripts, Depending on a Given Value

### Name

**switch** - Evaluate one of several scripts, depending on a given value

### Synopsis

**switch** *?options? string pattern body ?pattern body ...?*

**switch** *?options? string {pattern body ?pattern body ...?}*

### Description

The **switch** command matches its *string* argument against each of the *pattern* arguments in order. As soon as it finds a *pattern* that matches *string* it evaluates the following *body* argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last *pattern* argument is **default** then it matches anything. If no *pattern* argument matches *string* and no default is given, then the **switch** command returns an empty string.

If the initial arguments to **switch** start with - then they are treated as options.

The following options are currently supported:

#### **-exact**

Use exact matching when comparing *string* to a pattern. This is the default.

#### **-glob**

When matching *string* to the patterns, use glob-style matching (i.e. the same as implemented by the **string match** command).

--

Marks the end of options. The argument following this one will be treated as *string* even if it starts with a -.

Two syntaxes are provided for the *pattern* and *body* arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line switch commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *pattern* arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

If a *body* is specified as ``-" it means that the *body* for the next pattern should also be used as the body for this pattern (if the next pattern also has a body of ``-" then the body after that is used, and so on). This feature makes it possible to share a single *body* among several patterns.

Beware of how you place comments in **switch** commands. Comments should only be placed **inside** the execution body of one of the patterns, and not intermingled with the patterns.

Below are some examples of **switch** commands:

```
switch abc a - b {format 1} abc {format 2} default {format 3}
```

=> will return 2,

```
switch -regexp aaab {
    ^a.*b$ -
```

```
        b {format 1}
        a* {format 2}
        default {format 3}
    }
=> will return 1, and
    switch xyz {
        a -
        b {format 1}
        a* {format 2}
        default {format 3}
    }
=> will return 3.
    set nb_jambes 4
    switch $nb_jambes {
        2 {puts "Cela peut être un humain."}
        4 {puts "Cela peut être une vache."}
        6 {puts "Cela peut être une fourmis."}
        8 {puts "Cela peut être une araignée."}
        default {puts "Cela peut être n'importe quoi."}
    }
=> Cela peut être une vache.
```

### **See also**

**for, if**

## 2.61 **tell - Return Current Access Position for an Open Channel**

### Name

**tell** - Return current access position for an open channel

### Synopsis

**tell** *channelId*

### Description

Returns an integer string giving the current access position in *channelId*. This value returned is a byte offset that can be passed to **seek** in order to set the channel to a particular position. Note that this value is in terms of bytes, not characters like **read**.

The value returned is -1 for channels that do not support seeking.

*ChannelId* must be an identifier for an open channel such as a Tcl standard channel (**stdin**, **stdout**, or **stderr**), the return value from an invocation of **open** or **socket**, or the result of a channel creation command provided by a Tcl extension.

### See also

**file**, **open**, **close**, **gets**, **seek**

## 2.62 **time** - Time the Execution of a Script

### **Name**

**time** - Time the execution of a script

### **Synopsis**

**time** *script* *?count?*

### **Description**

This command will call the Tcl interpreter *count* times to evaluate *script* (or once if *count* isn't specified).

It will then return a string of the form

**503 microseconds per iteration**

which indicates the average amount of time required per iteration, in microseconds.

Time is measured in elapsed time, not CPU time.

### **See also**

**clock**

## 2.63 **trace - Monitor Variable Accesses, Command Usages and Command Executions**

### Name

**trace** - Monitor variable accesses, command usages and command executions

### Synopsis

**trace** *option* ?*arg arg ...?*

### Description

This command causes Tcl commands to be executed whenever certain operations are invoked. The legal *option*'s (which may be abbreviated) are:

**trace add** *type name ops ?args?*

Where *type* is command, execution, or variable.

**trace add** *command name ops command*

Arrange for *command* to be executed whenever command *name* is modified in one of the ways given by the list *ops*. *Name* will be resolved using the usual namespace resolution rules used by procedures. If the command does not exist, an error will be thrown. *Ops* indicates which operations are of interest, and is a list of one or more of the following items:

#### **rename**

Invoke *command* whenever the command is renamed. Note that renaming to the empty string is considered deletion, and will not be traced with '**rename**'.

#### **delete**

Invoke *command* when the command is deleted. Commands can be deleted explicitly by using the **rename** command to rename the command to an empty string. Commands are also deleted when the interpreter is deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, depending on the operations being traced, a number of arguments are appended to *command* so that the actual command is as follows:

*command oldName newName op*

*OldName* and *newName* give the traced command's current (old) name, and the name to which it is being renamed (the empty string if this is a 'delete' operation). *Op* indicates what operation is being performed on the command, and is one of **rename** or **delete** as defined above. The trace operation cannot be used to stop a command from being deleted. Tcl will always remove the command once the trace is complete. Recursive renaming or deleting will not cause further traces of the same type to be evaluated, so a delete trace which itself deletes the command, or a rename trace which itself renames the command will not cause further trace evaluations to occur. Both *oldName* and *newName* are fully qualified with any namespace(s) in which they appear.

**trace add** *execution name ops command*

Arrange for *command* to be executed whenever command *name* is executed, with traces occurring at the points indicated by the list *ops*. *Name* will be resolved using the usual namespace resolution rules used by procedures. If the command does not exist, an error will be thrown. *Ops* indicates which operations are of interest, and is a list of one or more of the following items:

#### **enter**

Invoke *command* whenever the command *name* is executed, just before the actual execution takes place.

**leave**

Invoke *command* whenever the command *name* is executed, just after the actual execution takes place.

**enterstep**

Invoke *command* for every tcl command which is executed inside the procedure *name*, just before the actual execution takes place. For example if we have 'proc foo { } { puts "hello" }', then a *enterstep* trace would be invoked just before *puts "hello"* is executed. Setting a *enterstep* trace on a *command* will not result in an error and is simply ignored.

**leavestep**

Invoke *command* for every tcl command which is executed inside the procedure *name*, just after the actual execution takes place. Setting a *leavestep* trace on a *command* will not result in an error and is simply ignored.

When the trace triggers, depending on the operations being traced, a number of arguments are appended to *command* so that the actual command is as follows:

For **enter** and **enterstep** operations:

*command command-string op*

*Command-string* gives the complete current command being executed (the traced command for a **enter** operation, an arbitrary command for a **enterstep** operation), including all arguments in their fully expanded form. *Op* indicates what operation is being performed on the command execution, and is one of **enter** or **enterstep** as defined above. The trace operation can be used to stop the command from executing, by deleting the command in question. Of course when the command is subsequently executed, an 'invalid command' error will occur.

For **leave** and **leavestep** operations:

*command command-string code result op*

*Command-string* gives the complete current command being executed (the traced command for a **enter** operation, an arbitrary command for a **enterstep** operation), including all arguments in their fully expanded form. *Code* gives the result code of that execution, and *result* the result string. *Op* indicates what operation is being performed on the command execution, and is one of **leave** or **leavestep** as defined above. Note that the creation of many **enterstep** or **leavestep** traces can lead to unintuitive results, since the invoked commands from one trace can themselves lead to further command invocations for other traces. *Command* executes in the same context as the code that invoked the traced operation: thus the *command*, if invoked from a procedure, will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *command* invokes a procedure (which it normally does) then the procedure will have to use *upvar* or *uplevel* commands if it wishes to access the local variables of the code which invoked the trace operation. While *command* is executing during an execution trace, traces on *name* are temporarily disabled. This allows the *command* to execute *name* in its body without invoking any other traces again. If an error occurs while executing the *command* body, then the command *name* as a whole will return that same error. When multiple traces are set on *name*, then for *enter* and *enterstep* operations, the traced commands are invoked in the reverse order of how the traces were originally created; and for *leave* and *leavestep* operations, the traced commands are invoked in the original order of creation. The behavior of execution traces is currently undefined for a command *name* imported into another namespace.

**trace add** *variable name ops command*

Arrange for *command* to be executed whenever variable *name* is accessed in one of the ways given by the list *ops*. *Name* may refer to a normal variable, an element of an array, or to an array as a whole (i.e. *name* may be just the name of an array, with no parenthesized index). If *name* refers to a whole array, then *command* is invoked

whenever any element of the array is manipulated. If the variable does not exist, it will be created but will not be given a value, so it will be visible to **namespace which** queries, but not to **info exists** queries. *Ops* indicates which operations are of interest, and is a list of one or more of the following items:

#### array

Invoke *command* whenever the variable is accessed or modified via the **array** command, provided that *name* is not a scalar variable at the time that the **array** command is invoked. If *name* is a scalar variable, the access via the **array** command will not trigger the trace.

#### read

Invoke *command* whenever the variable is read.

#### write

Invoke *command* whenever the variable is written.

#### unset

Invoke *command* whenever the variable is unset. Variables can be unset explicitly with the **unset** command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to *command* so that the actual command is as follows:

*command name1 name2 op*

*Name1* and *name2* give the name(s) for the variable being accessed: if the variable is a scalar then *name1* gives the variable's name and *name2* is an empty string; if the variable is an array element then *name1* gives the name of the array and *name2* gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then *name1* gives the array name and *name2* is an empty string. *Name1* and *name2* are not necessarily the same as the name used in the **trace variable** command: the **upvar** command allows a procedure to reference a variable under a different name. *Op* indicates what operation is being performed on the variable, and is one of **read**, **write**, or **unset** as defined above.

*Command* executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a Tcl procedure, then *command* will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *command* invokes a procedure (which it normally does) then the procedure will have to use **upvar** or **uplevel** if it wishes to access the traced variable. Note also that *name1* may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the **upvar** command.

For read and write traces, *command* can modify the variable to affect the result of the traced operation. If *command* modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from *command* is ignored except that if it returns an error of any sort then the traced operation also returns an error with the same error message returned by the trace command (this mechanism can be used to implement read-only variables, for example). For write traces, *command* is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, *command* will have to restore the old value of the variable.

While *command* is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by *command*

will occur directly, without invoking *command* (or any other traces) again. However, if *command* unsets the variable then unset traces will be invoked.

When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked. Any errors in unset traces are ignored.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the **trace remove variable** command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

This command returns an empty string.

**trace remove** type name opList command

Where type is either **command**, **execution** or **variable**.

**trace remove command** name opList command

If there is a trace set on command name with the operations and command given by opList and command, then the trace is removed, so that command will never again be invoked. Returns an empty string. If name doesn't exist, the command will throw an error.

**trace remove execution** name opList command

If there is a trace set on command name with the operations and command given by opList and command, then the trace is removed, so that command will never again be invoked. Returns an empty string. If name doesn't exist, the command will throw an error.

**trace remove variable** name opList command

If there is a trace set on variable name with the operations and command given by opList and command, then the trace is removed, so that command will never again be invoked. Returns an empty string.

**trace info** type name

Where type is either **command**, **execution** or **variable**.

**trace info command** name

Returns a list containing one element for each trace currently set on command name. Each element of the list is itself a list containing two elements, which are the opList and command associated with the trace. If name doesn't have any traces set, then the result of the command will be an empty string. If name doesn't exist, the command will throw an error.

**trace info execution** name

Returns a list containing one element for each trace currently set on command name. Each element of the list is itself a list containing two elements, which are the opList and command associated with the trace. If name doesn't have any traces set, then the result of the command will be an empty string. If name doesn't exist, the command will throw an error.

**trace info variable** name

Returns a list containing one element for each trace currently set on variable name. Each element of the list is itself a list containing two elements, which are the opList and command associated with the trace. If name doesn't exist or doesn't have any traces set, then the result of the command will be an empty string.

For backwards compatibility, three other subcommands are available:

**trace variable** name ops command

This is equivalent to **trace add variable** name ops command.

**trace vdelete** name ops command

This is equivalent to **trace remove variable** name ops command

**trace vinfo** name

This is equivalent to **trace info variable** name

These subcommands are deprecated and will likely be removed in a future version of Tcl. They use an older syntax in which **array**, **read**, **write**, **unset** are replaced by **a**, **r**, **w** and **u** respectively, and the ops argument is not a list, but simply a string concatenation of the operations, such as **rwua**.

**See also**

**set**, **unset**

## 2.64 **unset - Delete Variables**

### Name

**unset** - Delete variables

### Synopsis

**unset** *?-nocomplain? ?--? ?name name name ...?*

### Description

This command removes one or more variables. Each *name* is a variable name, specified in any of the ways acceptable to the **set** command. If a *name* refers to an element of an array then that element is removed without affecting the rest of the array. If a *name* consists of an array name with no parenthesized index, then the entire array is deleted.

The **unset** command returns an empty string as result. If *-nocomplain* is specified as the first argument, any possible errors are suppressed. The option may not be abbreviated, in order to disambiguate it from possible variable names. The option *--* indicates the end of the options, and should be used if you wish to remove a variable with the same name as any of the options. If an error occurs, any variables after the named one causing the error not deleted. An error can occur when the named variable doesn't exist, or the name refers to an array element but the variable is a scalar, or the name refers to a variable in a non-existent namespace.

### See also

**set**, **trace**

## 2.65 **update - Process Pending Events and Idle Callbacks**

### Name

**update** - Process pending events and idle callbacks

### Synopsis

**update** ?idletasks?

### Description

This command is used to bring the application "up to date" by entering the event loop repeatedly until all pending events (including idle callbacks) have been processed.

If the **idletasks** keyword is specified as an argument to the command, then no new events or errors are processed; only idle callbacks are invoked. This causes operations that are normally deferred, such as display updates, to be performed immediately.

The **update idletasks** command is useful in scripts where changes have been made to the application's state and you want those changes to appear on the display immediately, rather than waiting for the script to complete. Most display updates are performed as idle callbacks, so **update idletasks** will cause them to run. However, there are some kinds of updates that only happen in response to events; these updates will not occur in **update idletasks**.

The **update** command with no options is useful in scripts where you are performing a long-running computation but you still want the application to respond to events such as user interactions; if you occasionally call **update** then user input will be processed during the next call to **update**.

### See also

**after**

## 2.66 **uplevel - Execute a Script in a Different Stack Frame**

### Name

**uplevel** - Execute a script in a different stack frame

### Synopsis

**uplevel** *?level?* *arg ?arg ...?*

### Description

All of the *arg* arguments are concatenated as if they had been passed to **concat**; the result is then evaluated in the variable context indicated by *level*. **Uplevel** returns the result of that evaluation.

If *level* is an integer then it gives a distance (up the procedure calling stack) to move before executing the command. If *level* consists of # followed by a number then the number gives an absolute level number. If *level* is omitted then it defaults to **1**. *Level* cannot be defaulted if the first *command* argument starts with a digit or #.

For example, suppose that procedure **a** was invoked from top-level, and that it called **b**, and that **b** called **c**. Suppose that **c** invokes the **uplevel** command. If *level* is **1** or **#2** or omitted, then the command will be executed in the variable context of **b**. If *level* is **2** or **#1** then the command will be executed in the variable context of **a**. If *level* is **3** or **#0** then the command will be executed at top-level (only global variables will be visible).

The **uplevel** command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose **c** invokes the command

```
uplevel 1 {set x 43; d}
```

where **d** is another Tcl procedure. The **set** command will modify the variable **x** in **b**'s context, and **d** will execute at level 3, as if called from **b**.

If it in turn executes the command

```
uplevel {set x 42}
```

then the **set** command will modify the same variable **x** in **b**'s context: the procedure **c** does not appear to be on the call stack when **d** is executing. The command **``info level``** may be used to obtain the level of the current procedure.

**Uplevel** makes it possible to implement new control constructs as Tcl procedures (for example, **uplevel** could be used to implement the **while** construct as a Tcl procedure).

**namespace eval** is another way (besides procedure calls) that the Tcl naming context can change. It adds a call frame to the stack to represent the namespace context. This means each **namespace eval** command counts as another call level for **uplevel** and **upvar** commands. For example, **info level 1** will return a list describing a command that is either the outermost procedure call or the outermost **namespace eval** command. Also, **uplevel #0** evaluates a script at top-level in the outermost namespace (the global namespace).

### See also

**namespace**, **upvar**

## 2.67 **upvar** - Create Link to Variable in a Different Stack Frame

### Name

**upvar** - Create link to variable in a different stack frame

### Synopsis

**upvar** *?level?* *otherVar* *myVar* *?otherVar myVar ...?*

### Description

This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. *Level* may have any of the forms permitted for the **uplevel** command, and may be omitted if the first letter of the first *otherVar* isn't # or a digit (it defaults to 1).

For each *otherVar* argument, **upvar** makes the variable by that name in the procedure frame given by *level* (or at global level, if *level* is #0) accessible in the current procedure by the name given in the corresponding *myVar* argument.

The variable named by *otherVar* need not exist at the time of the call; it will be created the first time *myVar* is referenced, just like an ordinary variable. There must not exist a variable by the name *myVar* at the time **upvar** is invoked. *MyVar* is always treated as the name of a variable, not an array element. Even if the name looks like an array element, such as **a(b)**, a regular variable is created. *OtherVar* may refer to a scalar variable, an array, or an array element. **Upvar** returns an empty string.

The **upvar** command simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as Tcl procedures.

For example, consider the following procedure:

```
proc add2 name {
    upvar $name x
    set x [expr $x+2]
}
```

**Add2** is invoked with an argument giving the name of a variable, and it adds two to the value of that variable. Although **add2** could have been implemented using **uplevel** instead of **upvar**, **upvar** makes it simpler for **add2** to access the variable in the caller's procedure frame.

**namespace eval** is another way (besides procedure calls) that the Tcl naming context can change. It adds a call frame to the stack to represent the namespace context. This means each **namespace eval** command counts as another call level for **uplevel** and **upvar** commands. For example, **info level 1** will return a list describing a command that is either the outermost procedure call or the outermost **namespace eval** command. Also, **uplevel #0** evaluates a script at top-level in the outermost namespace (the global namespace).

If an upvar variable is unset (e.g. **x** in **add2** above), the **unset** operation affects the variable it is linked to, not the upvar variable. There is no way to unset an upvar variable except by exiting the procedure in which it is defined. However, it is possible to retarget an upvar variable by executing another **upvar** command.

### Traces and upvar

Upvar interacts with traces in a straightforward but possibly unexpected manner. If a variable trace is defined on *otherVar*, that trace will be triggered by actions involving *myVar*. However, the trace procedure will be passed the name of *myVar*, rather than the name of *otherVar*. Thus, the output of the following code will be **localVar** rather than **originalVar**:

```
proc traceproc { name index op } {
    puts $name
}
proc setByUpvar { name value } {
    upvar $name localVar
    set localVar $value
}

set originalVar 1
trace variable originalVar w traceproc
setByUpvar originalVar 2
}
```

If *otherVar* refers to an element of an array, then variable traces set for the entire array will not be invoked when *myVar* is accessed (but traces on the particular element will still be invoked). In particular, if the array is **env**, then changes made to *myVar* will not be passed to subprocesses correctly.

### See also

**global**, **namespace**, **uplevel**, **variable**

## 2.68 **variable** - Create and Initialize a Namespace Variable

### Name

**variable** - create and initialize a namespace variable

### Synopsis

**variable** *?name value...? name ?value?*

### Description

This command is normally used within a **namespace eval** command to create one or more variables within a namespace. Each variable *name* is initialized with *value*. The *value* for the last variable is optional.

If a variable *name* does not exist, it is created. In this case, if *value* is specified, it is assigned to the newly created variable. If no *value* is specified, the new variable is left undefined. If the variable already exists, it is set to *value* if *value* is specified or left unchanged if no *value* is given. Normally, *name* is unqualified (does not include the names of any containing namespaces), and the variable is created in the current namespace. If *name* includes any namespace qualifiers, the variable is created in the specified namespace. If the variable is not defined, it will be visible to the **namespace which** command, but not to the **info exists** command.

If the **variable** command is executed inside a Tcl procedure, it creates local variables linked to the corresponding namespace variables (and therefore these variables are listed by **info locals**.) In this way the **variable** command resembles the **global** command, although the **global** command only links to variables in the global namespace. If any *values* are given, they are used to modify the values of the associated namespace variables. If a namespace variable does not exist, it is created and optionally initialized.

A *name* argument cannot reference an element within an array. Instead, *name* should reference the entire array, and the initialization *value* should be left off. After the variable has been declared, elements within the array can be set using ordinary **set** or **array** commands.

### See also

**global**, **namespace**, **upvar**

## 2.69 **vwait - Process Events Until a Variable is Written**

### **Name**

**vwait** - Process events until a variable is written

### **Synopsis**

**vwait** *varName*

### **Description**

This command enters the Tcl event loop to process events, blocking the application if no events are ready. It continues processing events until some event handler sets the value of variable *varName*. Once *varName* has been set, the **vwait** command will return as soon as the event handler that modified *varName* completes. *varName* must globally scoped (either with a call to **global** for the *varName*, or with the full namespace path specification).

In some cases the **vwait** command may not return immediately after *varName* is set. This can happen if the event handler that sets *varName* does not complete immediately. For example, if an event handler sets *varName* and then itself calls **vwait** to wait for a different variable, then it may not return for a long time. During this time the top-level **vwait** is blocked waiting for the event handler to complete, so it cannot return either.

### **See also**

**global**

## 2.70 while - Execute Script Repeatedly as Long as a Condition is Met

### Name

**while** - Execute script repeatedly as long as a condition is met

### Synopsis

**while** *test body*

### Description

The **while** command evaluates *test* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must a proper boolean value; if it is a true value then *body* is executed by passing it to the Tcl interpreter. Once *body* has been executed then *test* is evaluated again, and the process repeats until eventually *test* evaluates to a false boolean value. Continue commands may be executed inside *body* to terminate the current iteration of the loop, and break commands may be executed inside *body* to cause immediate termination of the while command. The while command always returns an empty string.

---

### NOTE

***test* should almost always be enclosed in braces. If not, variable substitutions will be made before the while command starts executing, which means that variable changes made by the loop body will not be considered in the expression. This is likely to result in an infinite loop. If *test* is enclosed in braces, variable substitutions are delayed until the expression is evaluated (before each loop iteration), so changes in the variables will be visible.**

---

For an example, try the following script with and without the braces around **\$x<10**:

```
set x 0
while {$x<10} {
    puts "x is $x"
    incr x
}
```

### See also

**break, continue, for, foreach**

### 3.0 TCL commands Not Supported

---

**bgerror**  
**dde**  
**file** (*in case of: attributes, mtime, volume*)  
**filename**  
**history**  
**interp**  
**load**  
**memory**  
**msgcat**  
**package**  
**pkg::create**  
**pkg\_mkIndex**  
**re\_syntax**  
**regexp**  
**registry**  
**regsub**  
**resource**  
**tcltest**  
**tclvars**  
**unknown**

## 4.0 Boot Tcl Script

---

The boot TCL script is a program that starts automatically after the controller boot sequence. It gets defined in the *system.ini* configuration file in the GENERAL section.

```
[GENERAL]
```

```
BootScriptFileName = testarg.tcl
```

```
BootScriptArguments = arg1, arg2, arg3, arg4, arg5
```

*BootScriptFileName* is the file name of the TCL script. This file must be stored in the *..\Admin\Public\Scripts* folder of the XPS controller.

*BootScriptArguments* defines the list of arguments of the TCL script. The separator between two arguments is the comma.

Example:

A boot TCL script could for instance contain the initialization and home search of all motion groups. Once the controller finishes booting, the motion groups will automatically initialize and home.

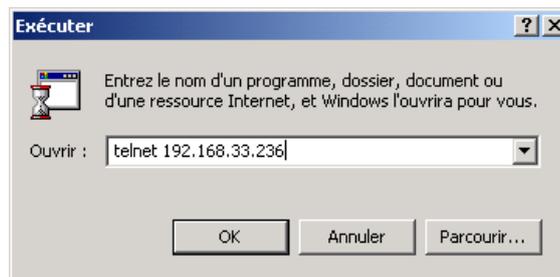
## 5.0 Telnet Connection

To follow the execution of a TCL script and to receive the messages and errors sent from the XPS controller, the use of a Telnet connection can be helpful. Telnet is a simple way to view the messages sent from the XPS to the *stdout*, but also to send data to the XPS.

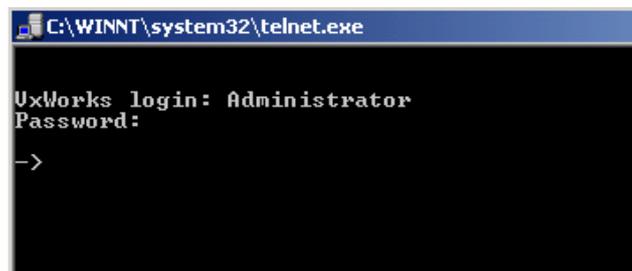
In TCL, the function *gets stdin* allows transmitting data from a Telnet window to a TCL script. However, there is no echo to the typed text, which means that users don't see the text that they enter.

A Telnet connection can be opened with any valid login, which can be administrator, anonymous, or whatever other logins are configured.

- For windows users, click *Start* -> *Run* -> then type *telnet* + IP address as below:



- The Telnet window is opened, type login (here login and password equal to "Administrator"):



- An arrow appears which indicates that the Telnet connection is ready to receive messages.

During the execution of TCL scripts, this window is the interface to the *stdout* and *stdin*. In the following example, the Telnet window displays the results of the TCL execution (it displays "Hello, World", gets the library and the firmware version).

```
# Display on console screen
puts stdout {Hello, World!}

# Get library version
set code [catch "GetLibraryVersion strVersion"]
if {$code != 0} {
    ErrorMessageGet $socketID $code strError
    puts "GetLibraryVersion Not OK => error = $code: $strError"
} else {
    puts stdout "Library Version = $strVersion"
}

# Open socket
set TimeOut 60
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
```

```

puts stdout "OpenConnection failed => $code"
} else {

# Get firmware version
set code [catch "FirmwareVersionGet $socketID strVersion"]
if {$code != 0} {
    ErrorStringGet $socketID $code strError
    puts "FirmwareVersionGet Not OK => error = $code:
$strError"
} else {
    puts stdout "Firmware Version = $strVersion"
}

# Close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
}

```

Sélectionner C:\WINNT\system32\telnet.exe

```

UxWorks login: Administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-> Hello, World!
Library Version = TCL_API_drivers for XPS-C8 Firmware U1.4.0
Firmware Version = XPS-C8 Firmware U1.4.0

```

## 6.0 Error Handling

For convenient error debugging and a safe program execution, the acknowledgements (errors) of each XPS command should be read and tested. The following script opens a TCP socket, and reads and displays the firmware version. If any error occurs, it gets and displays the description of the error before closing the TCP socket. The TCP socket opening and closing are tested as well.

```
# Initialization
set TimeOut 60
set ErrorCode 0

# Open TCP socket
set ErrorCode [catch "OpenConnection $TimeOut socketID"]
if {$ErrorCode != 0} {
    # Error => TCP socket not opened
    puts stdout "OpenConnection failed => $ErrorCode"
} else {

    # Success => Read firmware version
    set ErrorCode [catch "FirmwareVersionGet $socketID
FirmwareVersion"]
    if {$ErrorCode != 0} {
        # Error => Get error description
        set ErrorCode [catch "ErrorStringGet $socketID
$ErrorCode ErrorString"]
        if {$ErrorCode == 0} {
            # Display error description
            puts stdout "$ErrorCode: $ErrorString"
        }
    }
}

# Success => Display firmware version
puts stdout "Controller version is $FirmwareVersion"

# Close TCP socket
set ErrorCode [catch "TCP_CloseSocket $socketID"]
if {$ErrorCode != 0} {
    # Error
    puts stdout "TCP_CloseSocket failed => $ErrorCode"
} else {
    # Success
    puts stdout "The socket $socketID is closed "
}
}
```

A telnet connection (see chapter 5 *Telnet connection* for how to open a Telnet connection), allows to follow the execution of a TCL script. In this example there is no error, the socket 0 is opened, the installed firmware version is 1.4.0 and the socket 0 gets closed.

```

Sélectionner C:\WINNT\system32\telnet.exe
UxWorks login: Administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> Controller version is XPS-C8 Firmware U1.4.0
The socket 0 is closed

```

In the example above, the checking of the acknowledgments and the code to display errors in the telnet window is put after each API command. Alternative, a procedure “display error and close” can be used. This procedure gets defined at the beginning of the TCL scripts. In that case, users just have to call this procedure after each API. This allows a significant reduction of code when lots of API’s are used.

```

#####
##### Display error and close procedure #####
#####
proc DisplayErrorAndClose {socketID code APIName} {
# Set global variable
global tcl argv
# If error occurred other than Timeout error
if {$code != -2} {
# Error => Get error description
set code2 [catch "ErrorStringGet $socketID $code
strError"]

# If error occurred with the API ErrorStringGet
if {$code2 != 0} {

# Display API name, error code and ErrorStringGet error
code

# in the telnet window when using APIs
TCLScriptExecute or
# TCLScriptExecuteAndWait
puts "$APIName ERROR => $code / ErrorStringGet
ERROR => $code2"

# in the web terminal when using API
TCLScriptExecuteAndWait
set tcl_argv(0) "$APIName ERROR => $code"
} else {
# Display API name, number and description of the error
# in the telnet window when using APIs
TCLScriptExecute or
# TCLScriptExecuteAndWait
puts stdout "$APIName ERROR => $code: $strError"

# in the web terminal when using API
TCLScriptExecuteAndWait
set tcl_argv(0) "$APIName ERROR => $code:
$strError"
}
} else {
# Display Timeout error

```

```

        # in the telnet window when using APIs
TCLScriptExecute or
        # TCLScriptExecuteAndWait
        puts stdout "$APIName ERROR => $code: TCP timeout"
# in the web terminal when using API TCLScriptExecuteAndWait
        set tcl argv(0) "$APIName ERROR => $code: TCP timeout"
    }
    # Close TCP socket
    set code2 [catch "TCP_CloseSocket $socketID"]
    return
}
#####
##### Process #####
#####
# Initialization
set TimeOut 60
set ErrorCode 0
# Open TCP socket
set ErrorCode [catch "OpenConnection $TimeOut socketID"]
if {$ErrorCode != 0} {
# Error => TCP socket not opened
    puts stdout "TCP_ConnectToServer failed !"
} else {
# Success => Read firmware version
    set ErrorCode [catch "FirmwareVersionGet $socketID
FirmwareVersion"]
    if {$ErrorCode != 0} {
        # Error => Get error description
        DisplayErrorAndClose $socketID $code
        "FirmwareVersionGet"
        return
    }
    # Success => Display firmware version
    puts stdout "Controller version is $FirmwareVersion"
# Close TCP socket
    set ErrorCode [catch "TCP_CloseSocket $socketID"]
    if {$ErrorCode != 0} {
        # Error
        puts stdout "TCP_CloseSocket failed => $ErrorCode"
    } else {
        # Success
        puts stdout "The socket $socketID is closed "
    }
}
}

```

This way of error management is also used with the TCL scripts that get generated by the TCL generator, see Terminal of the XPS web interface. The procedure for displaying the errors and closing the TCP connection is as described above. And for each API the following code is used:

```

# Operation
    set ErrorCode [catch "GroupInitialize $socketID S"]
# Error management
    if {$ErrorCode != 0} {

```

```

        DisplayErrorAndClose $socketID $code
    "GroupInitialize"
        return
    }
}

```

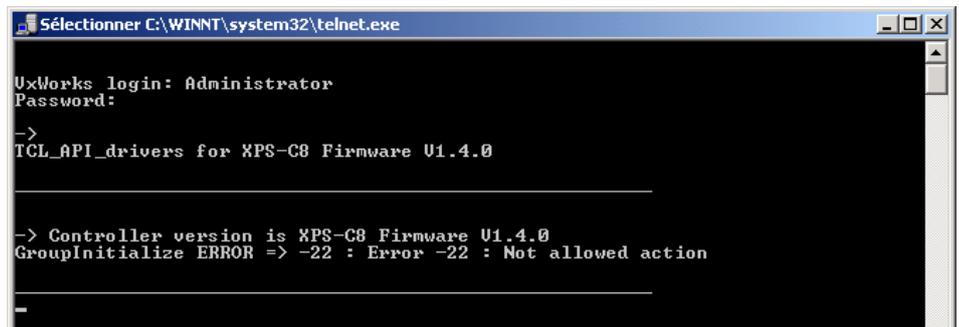
If an error occurs, it returns the first found error, indicates the API name that caused that error, and the number and the corresponding description of that error. The execution of the script gets stopped.

For instance, if we ask a group to initialize twice, it returns the following error:

```

# Open TCP socket
...
# Group initialization
    set code [catch "GroupInitialize $socketID S"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupInitialize"
        return
    }
    # Group initialization
    set code [catch "GroupInitialize $socketID S"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupInitialize"
        return
    }
# Close TCP socket
...

```



The screenshot shows a telnet window titled "Sélectionner C:\WINNT\system32\telnet.exe". The session starts with a login prompt for "UxWorks" with the username "Administrator". After the password is entered, the prompt changes to "->". The user enters "TCL\_API\_drivers for XPS-C8 Firmware U1.4.0", which is accepted. The prompt changes to "-> Controller version is XPS-C8 Firmware U1.4.0". The user then enters "GroupInitialize ERROR => -22 : Error -22 : Not allowed action".

## 7.0 Examples of Tcl Programs with XPS

Please refer to the XPS Programmer's Manual for the prototypes of the XPS API's when used from TCL.

### 7.1 Using analog I/O for Motion

#### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
XY	1	alignstation	alignstation.middle and alignstation.base

#### Description

This example opens a TCP connection, kills the XY group, then initializes and homes it. Five relative moves of 1 unit each are commanded to the group. Then, the value of the GPIO2 input is read in a continuous loop and sent to the *stdout* as long as the voltage of the analog input is above 0.2 volt. When above 0.2 volts, absolute moves are commanded to both axes: the X positioner moves corresponding to the voltage value of the analog input, and the Y positioner moves corresponding to the opposite of the voltage value of the analog input. When the GPIO2 input voltage is lower or equal to the limit of 0.2 volt, the last display and moves occurs. Finally the program ends by closing the socket.

If the voltage is below 0.2 volt already during the first reading, it directly goes to the end without displaying the I/O value or absolute moves of the XY group.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

#### TCL code

```
# Initialization
set TimeOut 10
set group "alignstation"
set axis1 "alignstation.middle"
set axis2 "alignstation.base"
set analogin "GPIO2.ADC1"

# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
    # Kill group
    set code [catch "GroupKill $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }
    # Initialize group
    set code [catch "GroupInitialize $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
        "GroupInitialize"
        return
    }
}
```

```

    # Home group
    set code [catch "GroupHomeSearch $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
    "GroupHomeSearch"
    return
}

# Move group with 5 relative units
for { set var 0 } { $var <= 5 } { incr var } {
    set code [catch "GroupMoveRelative $socketID
$group 1 1"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
    "GroupMoveRelative"
    return
}
}

# Get analog value
set code [catch "GPIOAnalogGet $socketID $analogin
voltage"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
}
"GPIOAnalogGet"
return
}

# Test if voltage is greater than 0.2 volt
while { $voltage >= 0.2 } {

    # Get analog value
    set code [catch "GPIOAnalogGet $socketID
$analogin voltage"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
    "GPIOAnalogGet"
    return
}
set move1 $voltage
set move2 [expr { $voltage * -1 }]
puts stdout "$analogin: $voltage volt(s)"
puts stdout "        move axis1: $move1"
puts stdout "        move axis2: $move2"
# Move axis 1
set code [catch "GroupMoveAbsolute $socketID
$axis1 $move1"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
}
"GroupMoveAbsolute"
return
}

# Move axis 2
set code [catch "GroupMoveAbsolute $socketID
$axis2 $move2"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
}
"GroupMoveAbsolute"
return
}
}
}

```

```

        return
    }
}
# Wait 1 second and close socket
after 1000
puts "End of program"
TCP CloseSocket $socketID
}

```

This is what gets displayed on a Telnet window. In this example the input voltage of GPIO2 decreases from 2.2V to 0V. See section 5 *Telnet connection* for details about Telnet connections:

```

C:\WINNT\system32\telnet.exe
UxWorks login: Administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware V1.4.0

-> GPIO2.ADC1: 2.201777002541 volt(s)
    move axis1: 2.20177700254
    move axis2: -2.20177700254
GPIO2.ADC1: 1.498232640273 volt(s)
    move axis1: 1.49823264027
    move axis2: -1.49823264027
GPIO2.ADC1: 1.572610022211 volt(s)
    move axis1: 1.57261002221
    move axis2: -1.57261002221
GPIO2.ADC1: 1.398248854716 volt(s)
    move axis1: 1.39824885472
    move axis2: -1.39824885472
GPIO2.ADC1: 0.8471187196984 volt(s)
    move axis1: 0.847118719698
    move axis2: -0.847118719698
GPIO2.ADC1: 0.450841520847 volt(s)
    move axis1: 0.450841520847
    move axis2: -0.450841520847
GPIO2.ADC1: -0.0283490611486 volt(s)
    move axis1: -0.0283490611486
    move axis2: 0.0283490611486
End of program

```

## 7.2 Using Digital I/O for Motion

### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
XY	1	alignstation	alignstation.middle and alignstation.base

### Description

This example opens a TCP connection, kills the XY group, then initializes and homes it. Five relative moves of 1 unit each are commanded to the group. Then, the value of the GPIO1 digital input is read in a continuous loop and sent to the *stdout* as long as the value of the input is different from 255. When the value of the digital GPIO1 input is equal to 1, absolute moves are commanded to both axes: the X positioner moves to the absolute position 1 and the Y positioner moves to the absolute position -1. When the value of 255 is obtained, the last display occurs. Finally, the program ends by closing the socket. If the GPIO1 input value is 255 already during the first reading, it directly goes to the end without displaying the digital input value.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### TCL Code

```
# Initialization
set Timeout 10
set group "alignstation"
set axis1 "alignstation.middle"
set axis2 "alignstation.base"
set digitalin "GPIO1.DI"
# Open TCP socket
set code [catch "OpenConnection $Timeout socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
    # Kill group
    set code [catch "GroupKill $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }
    # Initialize group
    set code [catch "GroupInitialize $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupInitialize"
        return
    }
    # Home group
    set code [catch "GroupHomeSearch $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupHomeSearch"
        return
    }
    # Move group with 5 relative units
    for { set var 0 } { $var <= 5 } { incr var } {
        set code [catch "GroupMoveRelative $socketID $group 1 1"]
        if {$code != 0} {
```

```

        DisplayErrorAndClose $socketID $code
    "GroupMoveRelative"
        return
    }
}
# Get digital value
set code [catch "GPIODigitalGet $socketID $digitalin value"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalGet"
    return
}
# Test if value of GPIO1.DI is different from 255
while { $value != 255 } {

    # Get digital value
    set code [catch "GPIODigitalGet $socketID $digitalin
value"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GPIODigitalGet"
        return
    }

    puts stdout "$digitalin: $value"
    if { $value == 1 } {
        puts "          move axis1: 1"
        puts "          move axis2: -1"

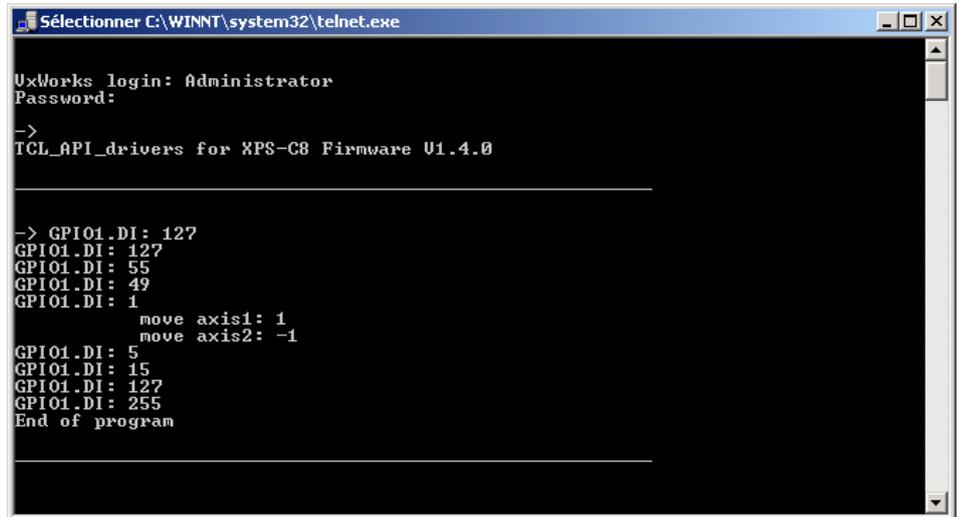
        # Move axis 1
        set code [catch "GroupMoveAbsolute $socketID $axis1 1"]
        if {$code != 0} {
            DisplayErrorAndClose $socketID $code
        }
    "GroupMoveAbsolute"
        return
    }

    # Move axis 2
    set code [catch "GroupMoveAbsolute $socketID $axis2 -
1"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
    "GroupMoveAbsolute"
        return
    }
    else {
        after 100
    }
    after 1000
}

# Wait 1 second and close socket
after 1000
puts "End of program"
TCP_CloseSocket $socketID
}

```

This is what gets displayed on a Telnet window. See section 5 *Telnet connection* for details about Telnet connections:



```
Sélectionner C:\WINNT\system32\telnet.exe

UxWorks login: administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> GPI01.DI: 127
GPI01.DI: 127
GPI01.DI: 55
GPI01.DI: 49
GPI01.DI: 1
      move axis1: 1
      move axis2: -1
GPI01.DI: 5
GPI01.DI: 15
GPI01.DI: 127
GPI01.DI: 255
End of program

-----
```

## 7.3 Test GPIO1

### Description

This example opens a TCP connection. It sets the value of 255 to the mask and the output GPIO1.DO, then gets this output value and puts it in the variable OA. It sets the value of 255 to the mask and the value of 0 to the output GPIO1.DO, then gets this output value and puts it in the variables OB. It sets the value of 63 to the mask and the value of 255 to the output GPIO1.DO, then gets this output value and puts it in the variable OC. After the settings, it tests the contents of the variables OA, OB and OC. Finally, the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

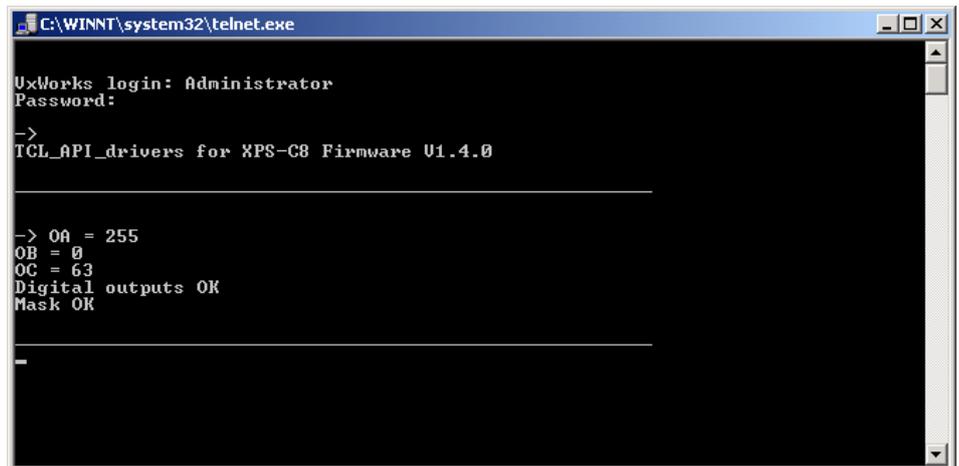
```
# Initialization
set TimeOut 120
set output "GPIO1.DO"
set input "GPIO1.DI"
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Set output of GPIO1 to 255
set code [catch "GPIODigitalSet $socketID $output 255 255"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalSet"
    return
}
# Get value of output of GPIO1 and store it in OA
set code [catch "GPIODigitalGet $socketID $output OA"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalGet"
    return
} else {
    puts "OA = $OA"
}
# Set output of GPIO1 to 0
set code [catch "GPIODigitalSet $socketID $output 255 0"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalSet"
    return
}
# Get value of output of GPIO1 and store it in OB
set code [catch "GPIODigitalGet $socketID $output OB"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalGet"
    return
} else {
    puts "OB = $OB"
}
# Set output of GPIO1 to 63 (mask value)
```

```

set code [catch "GPIODigitalSet $socketID $output 63 255"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalSet"
    return
}
# Get value of output of GPIO1 and store it in OC
set code [catch "GPIODigitalGet $socketID $output OC"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalGet"
    return
} else {
    puts "OC = $OC"
}
# Test if OA = 255 and OB = 0
if {$OA == 255 & $OB == 0} {
    puts "Digital outputs OK"
}
# Test if OC = 63
if {$OC == 63} {
    puts "Mask OK"
} else {
    puts "Pb Mask"
}
} else {
    puts "Pb digital outputs"
}
}
# Close socket
TCP_CloseSocket $socketID
}

```

This is what gets displayed on a Telnet window for the above example. For details about Telnet connections, see section 5 *Telnet connection*:



The screenshot shows a Telnet window titled "C:\WINNT\system32\telnet.exe". The window displays the following text:

```

UxWorks login: Administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> OA = 255
OB = 0
OC = 63
Digital outputs OK
Mask OK

-----

```

## 7.4 Gathering with Motion

### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

### Description

This example opens a TCP connection, kills the single axis group, then initializes and homes it. Then, it configures the parameters for the gathering (data to be collected: setpoint and current positions). It defines an action (GatheringRun) to an event (SGamma.MotionStart). When the positioner moves from 0 to 50, the data are gathered (with a divisor equal to 100, data are collected every 100<sup>th</sup> servo cycle, or every 10 ms). At the end, the gathering is stopped and saved in a text file (*Gathering.dat* in Admin/Public directory of the controller). Finally, the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

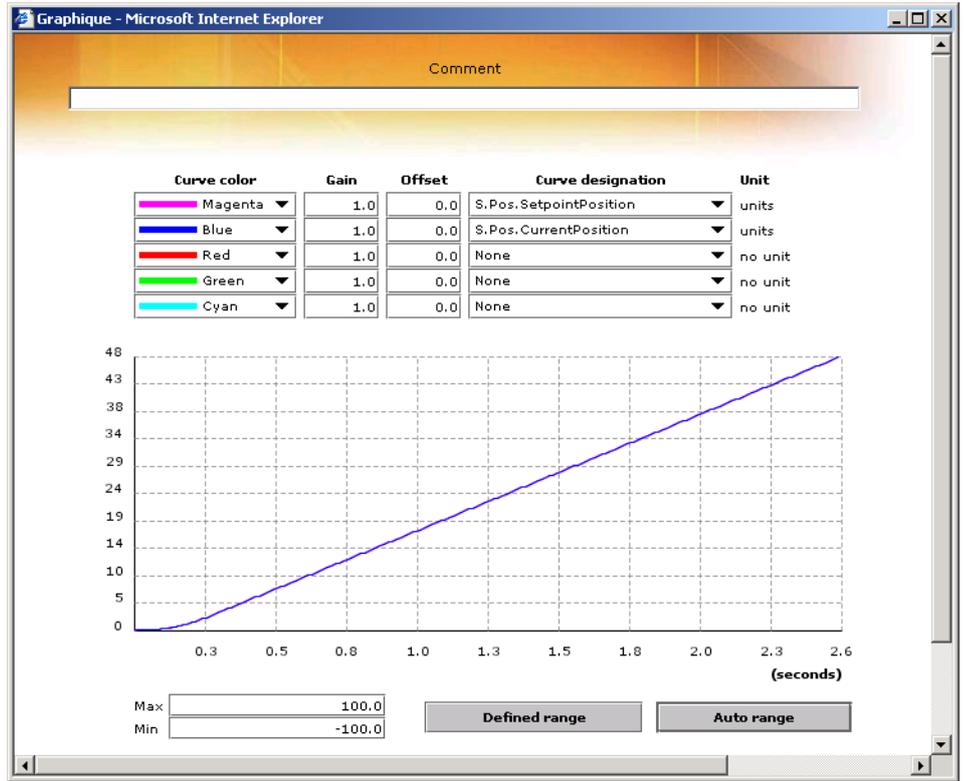
### Code

```
# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE_AXIS.MY_STAGE"
set Type1 "SINGLE_AXIS.MY_STAGE.SetpointPosition"
set Type2 "SINGLE_AXIS.MY_STAGE.CurrentPosition"
set Event "SGamma.MotionStart"
set Action "GatheringRun"
set Displacement 50
set NbPoints 1000
set Div 100
set code 0

# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Kill group
    set code [catch "GroupKill $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }
# Initialize group
    set code [catch "GroupInitialize $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupInitialize"
        return
    }
# Home group
    set code [catch "GroupHomeSearch $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupHomeSearch"
    }
}
```

```
        return
    }
}
# Configure gathering parameters
set code [catch "GatheringConfigurationSet $socketID $Type1
$Type2"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringConfigurationSet"
    return
}
# Add an event
set code [catch "EventAdd $socketID $Positioner $Event 0
$Action $NbPoints $Div 0"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "EventAdd"
    return
}
# Move positioner
set code [catch "GroupMoveRelative $socketID $Group
$Displacement"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupMoveRelative"
    return
}
# Stop gathering and save data
set code [catch "GatheringStopAndSave $socketID"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringStopAndSave"
    return
}
# Close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
}
```

When pressing the *Gathering display* button of Terminal window of the XPS web site interface, the following data gets displayed:



## 7.5 External Gathering

### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

### Description

This example opens a TCP connection, kills the single axis group, then initializes and homes it. Then, it configures the parameters for the external gathering (data to be collected: ExternalLatchPosition and GPIO2.ADC1 value). It defines an action (ExternalGatheringRun) to an event (Immediate). Each time the trigger in receives a signal, the data is gathered (with a divisor equal to 1, gathering takes place every signal on the trigger input). Every second, the current number of gathered data gets displayed. At the end, the external gathering is stopped and saved in a text file (*ExternalGathering.dat* in Admin/Public directory of the controller). Finally, the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE_AXIS.MY_STAGE"
set Type1 "SINGLE_AXIS.MY_STAGE.ExternalLatchPosition"
set Type2 "GPIO2.ADC1"
set Event "Immediate"
set Action "ExternalGatheringRun"
set NbPoints 20
set Div 1
set Current 0
set code 0

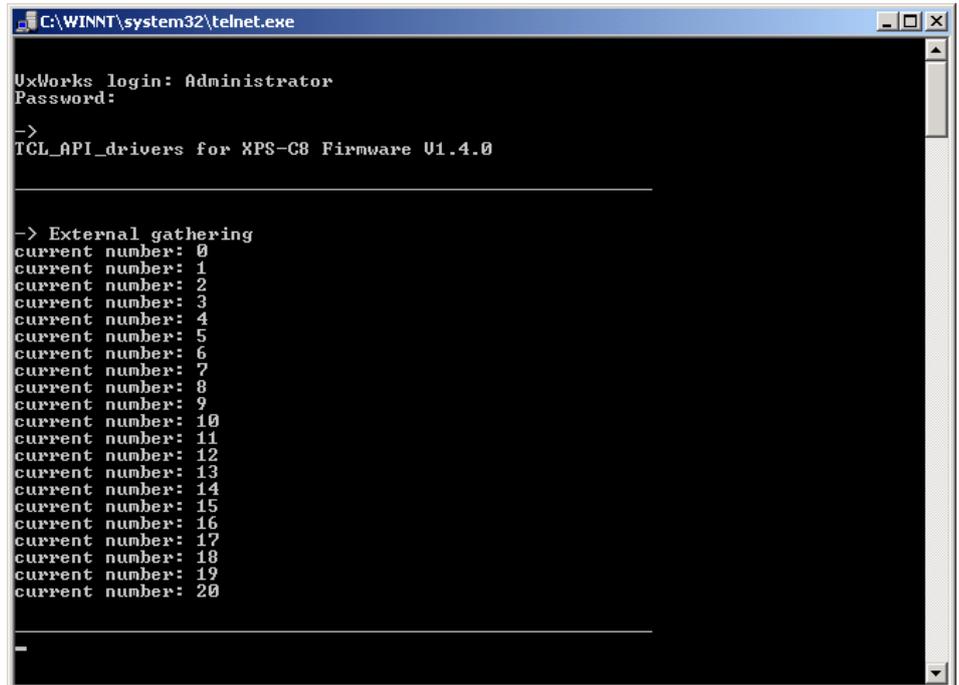
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Kill group
    set code [catch "GroupKill $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }
# Initialize group
    set code [catch "GroupInitialize $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupInitialize"
        return
    }
# Home group
    set code [catch "GroupHomeSearch $socketID $Group"]
    if {$code != 0} {
```

```

    DisplayErrorAndClose $socketID $code "GroupHomeSearch"
    return
}
# Configure gathering parameters
set code [catch "GatheringExternalConfigurationSet $socketID
$Type1 $Type2"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringExternalConfigurationSet"
    return
}
# Add an event
set code [catch "EventAdd $socketID $Positioner $Event 0
$Action $NbPoints $Div 0"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "EventAdd"
    return
}
# Push on TRIG IN button...
puts "External gathering"
# Wait end of external gathering
while {$Current < $NbPoints} {
# Get current acquired point number
set code [catch "GatheringExternalCurrentNumberGet
$socketID Current Max"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringExternalCurrentNumberGet"
    return
} else {
    puts stdout "current number: $Current"
    after 1000
}
}
# Stop external gathering and save data
set code [catch "GatheringExternalStopAndSave $socketID"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringExternalStopAndSave"
    return
}
# Close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
}

```

This is what gets displayed on a Telnet window for the above example and when the trigger in receives a signal every second. For details about Telnet connections, see section 5 *Telnet connection*:



```
C:\WINNT\system32\telnet.exe
UxWorks login: administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> External gathering
current number: 0
current number: 1
current number: 2
current number: 3
current number: 4
current number: 5
current number: 6
current number: 7
current number: 8
current number: 9
current number: 10
current number: 11
current number: 12
current number: 13
current number: 14
current number: 15
current number: 16
current number: 17
current number: 18
current number: 19
current number: 20

-----
```

## 7.6 Position Compare

### Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

### Description

This example opens a TCP connection, kills the single axis group, then initializes and homes it. With an absolute move, the positioner moves to the start position  $-15$ . Then, it configures the parameters for the position compare (enabled from  $-10$  to  $+10$  with step position of 1 unit). It enables the position compare functionality and executes a relative move of 25 (positioner final position will be  $-15+25 = +10$ ). During this move, between the positions  $-10$  and  $+10$ , pulses are sent by the trigger output when crossing each 1 unit incremental position. The position compare mode is then disabled and the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE_AXIS.MY_STAGE"
set StartPosition -15
set Displacement 25
set MinPos -10
set MaxPos 10
set StepPos 1
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Kill group
set code [catch "GroupKill $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupKill"
    return
}
# Initialize group
set code [catch "GroupInitialize $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupInitialize"
    return
}
# Home group
set code [catch "GroupHomeSearch $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupHomeSearch"
    return
}
}
```

```

# Move positioner to start position
    set code [catch "GroupMoveAbsolute $socketID $Group
$StartPosition"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupMoveAbsolute"
        return
    }
# Set position compare parameters
    set code [catch "PositionerPositionCompareSet $socketID
$Positioner $MinPos $MaxPos $StepPos"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
        "PositionerPositionCompareSet"
        return
    }
# Enable position compare mode
    set code [catch "PositionerPositionCompareEnable $socketID
$Positioner"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
        "PositionerPositionCompareEnable"
        return
    }
# Move positioner
    set code [catch "GroupMoveRelative $socketID $Group
$Displacement"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupMoveRelative"
        return
    }
# Disable position compare mode
    set code [catch "PositionerPositionCompareDisable $socketID
$Positioner"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
        "PositionerPositionCompareDisable"
        return
    }
# Close TCP socket
    set code [catch "TCP_CloseSocket $socketID"]
}

```

## 7.7 Master-Slave Mode

### Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE
XY	1	XY	XY.X and XY.Y

### Description:

This example opens a TCP connection, kills the single axis and the XY group, then initializes and homes them. It sets the parameters for the master slave mode (slave: single axis group, master: X positioner from XY group). Then, it enables the master slave mode and executes a relative move of 65 units with the master positioner. At the same time, the slave positioner executes the same move as the master. The master slave mode is then disabled and the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set Timeout 60
set SlaveGroup "SINGLE_AXIS"
set XYGroup "XY"
set MasterPositioner "XY.X"
set MasterRatio 1
set code 0
set Displacement 65
# Open TCP socket
set code [catch "OpenConnection $Timeout socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
    # Kill single axis group
    set code [catch "GroupKill $socketID $SlaveGroup"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "Single axis
GroupKill"
        return
    }
    # Initialize single axis group
    set code [catch "GroupInitialize $socketID $SlaveGroup"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "Single axis
GroupInitialize"
        return
    }
    # Home single axis group
    set code [catch "GroupHomeSearch $socketID $SlaveGroup"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "Single axis
GroupHomeSearch"
        return
    }
    # Kill XY group
```

```

set code [catch "GroupKill $socketID $XYGroup"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "XY GroupKill"
    return
}

# Initialize XY group
set code [catch "GroupInitialize $socketID $XYGroup"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "XY GroupInitialize"
    return
}

# Home XY group
set code [catch "GroupHomeSearch $socketID $XYGroup"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "XY GroupHomeSearch"
    return
}

# Set slave (single axis group) with its master
# (positioner from any group: XY here)
set code [catch "SingleAxisSlaveParametersSet $socketID
$SlaveGroup $MasterPositioner $MasterRatio"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
    "SingleAxisSlaveParametersSet"
    return
}

# Enable master-slave mode (group must be ready)
set code [catch "SingleAxisSlaveModeEnable $socketID
$SlaveGroup"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
    "SingleAxisSlaveModeEnable"
    return
}

# Move master positioner
# (the slave must follow the master in relation to a ratio)
set code [catch "GroupMoveRelative $socketID
$MasterPositioner $Displacement"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupMoveRelative"
    return
}

# Disable master-slave mode
set code [catch "SingleAxisSlaveModeDisable $socketID
$SlaveGroup"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
    "SingleAxisSlaveModeDisable"
    return
}

# Close TCP socket
set code [catch "TCP CloseSocket $socketID"]
}

```

## 7.8 Jogging

### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
XY	1	XY	XY.X and XY.Y

### Description

This example opens a TCP connection, kills the XY group, then initializes and homes it. It enables the jog mode and sets the parameters to move a positioner in the positive direction with a velocity of 20 units/s during 3 seconds. Then, during the 3 next seconds, the positioner moves in the reverse direction with a velocity of -30 units/s, and finally stops (velocity set to 0). The jog functionality is then disabled and the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set TimeOut 60
set Group "XY"
set Positioner "XY.X"
set Velocity1 20
set Velocity2 -30
set Acceleration 80
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Kill group
set code [catch "GroupKill $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupKill"
    return
}
# Initialize group
set code [catch "GroupInitialize $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupInitialize"
    return
}
# Home group
set code [catch "GroupHomeSearch $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupHomeSearch"
    return
}
# Enable jog mode (group must be ready)
set code [catch "GroupJogModeEnable $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupJogModeEnable"
```

```

        return
    }
}
# Set jog parameters to move a positioner => constant velocity
is not null
    set code [catch "GroupJogParametersSet $socketID $Positioner
$Velocity1 $Acceleration"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
"GroupJogParametersSet"
    return
}
# Wait 3 seconds
after 3000
# Set jog parameters to move the positioner in the reverse
sense
# => constant velocity is not null
    set code [catch "GroupJogParametersSet $socketID $Positioner
$Velocity2 $Acceleration"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
"GroupJogParametersSet"
    return
}
# Wait 3 seconds
after 3000
# Set jog parameters to stop a positioner => constant
velocity is null
    set code [catch "GroupJogParametersSet $socketID $Positioner
0 $Acceleration"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
    }
"GroupJogParametersSet"
    return
}
# Disable jog mode
# (constant velocity must be null on all positioners from
group)
    set code [catch "GroupJogModeDisable $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupJogModeDisable"
    }
    return
}
# Close TCP socket
    set code [catch "TCP_CloseSocket $socketID"]
}

```

## 7.9 Jogging and Gathering

### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
XY	1	XY	XY.X_VP and XY.Y_VP

### Description

This example opens a TCP connection, kills the XY group, then initializes and homes it. Then, it configures the parameters for the gathering (data to be collected: setpoint position, current position, setpoint velocity and setpoint acceleration). It displays the maximum number of acquisition per type of data that can be collected (max total data acquisition/number of data types = 1000000/4 = 250000). It defines an action (GatheringRun) to an event (Immediate). When the jog mode is enabled, it changes the jogging speed and acceleration. At the end, the jog mode is disabled, the gathering is stopped and saved in a text file (*Gathering.dat* in Admin/Public directory of the controller). Finally, the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set TimeOut 120
set Moteur "XY"
set Mot "XY.X_VP"
set A "XY.X_VP.SetpointPosition"
set B "XY.X_VP.CurrentPosition"
set C "XY.X_VP.SetpointVelocity"
set D "XY.X_VP.SetpointAcceleration"
set Event "Immediate"
set Action "GatheringRun"
set Num 0

# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Kill group
set code [catch "GroupKill $socketID $Moteur"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupKill"
    return
}
# Initialize group
set code [catch "GroupInitialize $socketID $Moteur"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupInitialize"
    return
}
# Home group
set code [catch "GroupHomeSearch $socketID $Moteur"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupHomeSearch"
}
```

```

        return
    }
}
# Set gathering parameters
set code [catch "GatheringConfigurationSet" $socketID $A $B
$C $D"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringConfigurationSet"
    return
}
# Get gathering parameters
set code [catch "GatheringConfigurationGet" $socketID J"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringConfigurationGet"
    return
} else {
    puts stdout "Data types to be gathered: $J"
}
}

# Get gathering current acquired point number
set code [catch "GatheringCurrentNumberGet" $socketID Num
Max"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringCurrentNumberGet"
    return
} else {
    puts stdout "Maximum possible number of acquisition per
type of data: $Max"
}
}

# Add an event
set code [catch "EventAdd" $socketID $Mot $Event 0 $Action
20000 10 0"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "EventAdd"
    return
}
}

# Enable jog mode
set code [catch "GroupJogModeEnable" $socketID $Moteur"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupJogModeEnable"
    return
}
}

# Wait 2 seconds
after 2000
puts "Jog moves and data acquisition"
# Set jog parameters to move both positioners in the
positive direction
set code [catch "GroupJogParametersSet" $socketID $Moteur 5
50 5 50"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupJogParametersSet"
    return
}
}

```

```

}
puts " X and Y positioners going in positive direction
during 500 msec"
puts "      X and Y positioners speed = 5 / Acceleration =
50"
# Wait 500 milliseconds
after 500
# Set jog parameters to move both positioners,
# the first in the positive direction, the second in the
negative
set code [catch "GroupJogParametersSet $socketID $Moteur 10
50 -10 50"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupJogParametersSet"
    return
}
puts " X positioner going in positive direction during 2
sec"
puts " Y positioner going in negative direction during 2
sec"
puts "      X positioner speed = 10 / Acceleration = 50"
puts "      Y positioner speed = -10 / Acceleration = 50"
# Wait 2 seconds
after 2000
# Set jog parameters to move both positioners in the reverse
sense
set code [catch "GroupJogParametersSet $socketID $Moteur -10
50 20 50"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupJogParametersSet"
    return
}
puts " X positioner going in negative direction during 2
sec"
puts " Y positioner going in positive direction during 2
sec"
puts "      X positioner speed = -10 / Acceleration = 50"
puts "      Y positioner speed = 20 / Acceleration = 50"
# Wait 2 seconds
after 2000
# Set jog parameters to move both positioners in the
negative direction
set code [catch "GroupJogParametersSet $socketID $Moteur -5
50 -5 50"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupJogParametersSet"
    return
}
puts " X and Y positioners going in negative direction
during 500 msec"
puts "      X and Y positioner speed = -5 / Acceleration = 50"
# Wait 500 milliseconds
after 500

```

```

# Set jog parameters to stop the positioners => constant
velocities are null
set code [catch "GroupJogParametersSet $socketID $Moteur 0
50 0 50"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupJogParametersSet"
    return
}
puts " X and Y positioners stopped"
puts " X and Y positioner speed = 0 / Acceleration =
50"
# Wait 500 milliseconds
after 500
# Disable jog mode
set code [catch "GroupJogModeDisable $socketID $Moteur"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupJogModeDisable"
    return
}
# Stop gathering and save data
set code [catch "GatheringStopAndSave $socketID"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GatheringStopAndSave"
    return
}
# Close socket
puts "End of program"
TCP_CloseSocket $socketID
}

```

This is what gets displayed on a Telnet window for the above example. For details about Telnet connections, see section 5 *Telnet connection*:

```

C:\WINNT\system32\telnet.exe
UxWorks login: Administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-> Data types to be gathered: XY.X.SetpointPosition;XY.X.CurrentPosition;XY.X.SetpointVelocity;XY.X.SetpointAcceleration
Maximum possible number of acquisition per type of data: 250000
Jog moves and data acquisition
X and Y positioners going in positive direction during 500 msec
X and Y positioners speed = 5 / Acceleration = 50
X positioner going in positive direction during 2 sec
Y positioner going in negative direction during 2 sec
X positioner speed = 10 / Acceleration = 50
Y positioner speed = -10 / Acceleration = 50
X positioner going in negative direction during 2 sec
Y positioner going in positive direction during 2 sec
X positioner speed = -10 / Acceleration = 50
Y positioner speed = 20 / Acceleration = 50
X and Y positioners going in negative direction during 500 msec
X and Y positioner speed = -5 / Acceleration = 50
X and Y positioners stopped
X and Y positioner speed = 0 / Acceleration = 50
End of program

```

## 7.10 Analog Position Tracking

### Configuration

Group type	Number	Group name	Positioner name
XY	1	XY	XY.X and XY.Y

### Description

This example opens a TCP connection, kills the XY group, then initializes and homes it. It sets the parameters for the position analog tracking functionality (positioner, analog input, offset, scale, velocity and acceleration) and enables the analog tracking mode. The mode gets activated during 20 seconds. During this time, the stage follows in position the voltage of the analog input GPIO2.ADC1. Then, the analog tracking mode gets disabled and the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set TimeOut 60
set Group "XY"
set Positioner "XY.X"
set AnalogInput "GPIO2.ADC1"
set Offset 0
set Scale 1
set Velocity 20
set Acceleration 80
set TrackingType "Position"
set code 0

# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
    # Kill group
    set code [catch "GroupKill $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }
    # Initialize group
    set code [catch "GroupInitialize $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupInitialize"
        return
    }
    # Home group
    set code [catch "GroupHomeSearch $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupHomeSearch"
        return
    }
    # Set analog tracking parameters
```

```

    set code [catch
"PositionerAnalogTrackingPositionParametersSet $socketID
$Positioner $AnalogInput $Offset $Scale $Velocity
$Acceleration"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code
        "PositionerAnalogTrackingPositionParametersSet"
        return
    }
}
# Enable analog position tracking mode (group must be ready)
set code [catch "GroupAnalogTrackingModeEnable $socketID
$Group $TrackingType"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupAnalogTrackingModeEnable"
    return
}
# Change the amplitude of GPIO2.ADC1 analog input during 20
seconds
after 20000
# Disable analog position tracking mode
set code [catch "GroupAnalogTrackingModeDisable $socketID
$Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"GroupAnalogTrackingModeDisable"
    return
}
}
# Close TCP socket
set code [catch "TCP CloseSocket $socketID"]
}

```

## 7.11 Backlash Compensation

### Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

### Description

This example opens a TCP connection and kills the single axis group. It enables the backlash compensation capability (for this the controller must be in the not initialized state). The group gets then initialized and homed. The value of the backlash compensation is set to 0.1. The positioner executes relative moves with the backlash compensation. Finally, the backlash compensation gets disabled and the program ends by closing the socket.

#### CAUTION



- The *HomeSearchSequenceType* in the *stages.ini* file must be different from *CurrentPositionAsHome*.
- The *Backlash* parameter in the *stages.ini* file must be greater than zero.
- To apply any modifications of the *stages.ini*, the controller must be rebooted.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set Timeout 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE_AXIS.MY_STAGE"
set BacklashValue 0.1
set Displacement 10
set code 0
# Open TCP socket
set code [catch "OpenConnection $Timeout socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {

    # Kill group
    set code [catch "GroupKill $socketID $Group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }

    # Enable backlash compensation

#####
# CAUTION: #
# Group must be "not initialized" and Backlash>0 in the #
```

```

# "stages.ini" file #
#####
set code [catch "PositionerBacklashEnable $socketID
$Positioner"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"PositionerBacklashEnable"
    return
}
# Initialize group
set code [catch "GroupInitialize $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupInitialize"
    return
}
# Home group
set code [catch "GroupHomeSearch $socketID $Group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupHomeSearch"
    return
}
# Modify Backlash value
# Caution: Backlash > 0 in the file "stages.ini"
set code [catch "PositionerBacklashSet $socketID $Positioner
$BacklashValue"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"PositionerBacklashSet"
    return
}
# Move group in positive direction
set code [catch "GroupMoveRelative $socketID $Group
$Displacement"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupMoveRelative"
    return
}
# Move group in negative direction
set code [catch "GroupMoveRelative $socketID $Group -
$Displacement"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupMoveRelative"
    return
}
# Disable Backlash (if you want to do trajectory, jogging or
tracking)
# CAUTION: to enable backlash, you must call "GroupKill" or
"KillAll" to
# come back in "not initialized" status
set code [catch "PositionerBacklashDisable $socketID
$Positioner"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code
"PositionerBacklashDisable"
    return
}

```

```
}  
# Close TCP socket  
set code [catch "TCP_CloseSocket $socketID"]  
}
```

## 7.12 Timer Event and Global Variables

### Configuration

<i>Group type</i>	<i>Number</i>	<i>Group name</i>	<i>Positioner name</i>
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

### Description

The script *StartScript.tcl* opens a TCP connection, configures a timer and uses this timer as an event. The action, in relation to this timer event, executes a second TCL script named *MyScript.tcl*. The *StartScript.tcl* script sets a global variable and closes the socket.

The timer is a permanent event. The frequency of the timer is set by the divisor, in this example 20000, which means that the second TCL script gets executed every 20000<sup>th</sup> servo loop or every 2 seconds (divisor/servo loop rate = 20000/10000 = 2 seconds).

The script *MyScript.tcl* reads the global variable, increments it as long as the variable is below 10. When the global variable is equal to 10, the second script deletes the timer event and finally, the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

- *StartScript.tcl*

```
# Initialization
set TCPTimeOut 0.5
set code 0
set ISRPeriodSec 0.0001
set Positioner "SINGLE_AXIS.MY_STAGE"
set TimerName "Timer1"
set TimerPeriodSec 2
set EvtParam 0
set Action "ExecuteTCLScript"
set TCLFile "MyScript.tcl"
set TCLTask "MyTask"
set TCLArgs "0"
set GlobalVarNumber 1
set Value 5
# Open TCP socket
set code [catch "OpenConnection $TCPTimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Calculate divisor (periods are in seconds)
    set Divisor [expr {$TimerPeriodSec / $ISRPeriodSec}]
    puts stdout "Divisor: $Divisor"
    set Divisor [expr {int($Divisor)}]
    puts stdout "Divisor troncated in integer: $Divisor"
# Configure a timer
    set code [catch "TimerSet $socketID $TimerName $Divisor"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "TimerSet"
    }
    return
}
```

```

} else {
    puts "Timer set"
}
# Add timer event with an action that allows to execute
"MyScript.tcl"
set code [catch "EventAdd $socketID $Positioner $TimerName
$EvtParam $Action $TCLFile $TCLTask $TCLArgs"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "EventAdd"
    return
}
# Set global variable
set code [catch "GlobalArraySet $socketID $GlobalVarNumber
$Value"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GlobalArraySet"
    return
}
# close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
}

```

- *MyScript.tcl*

```

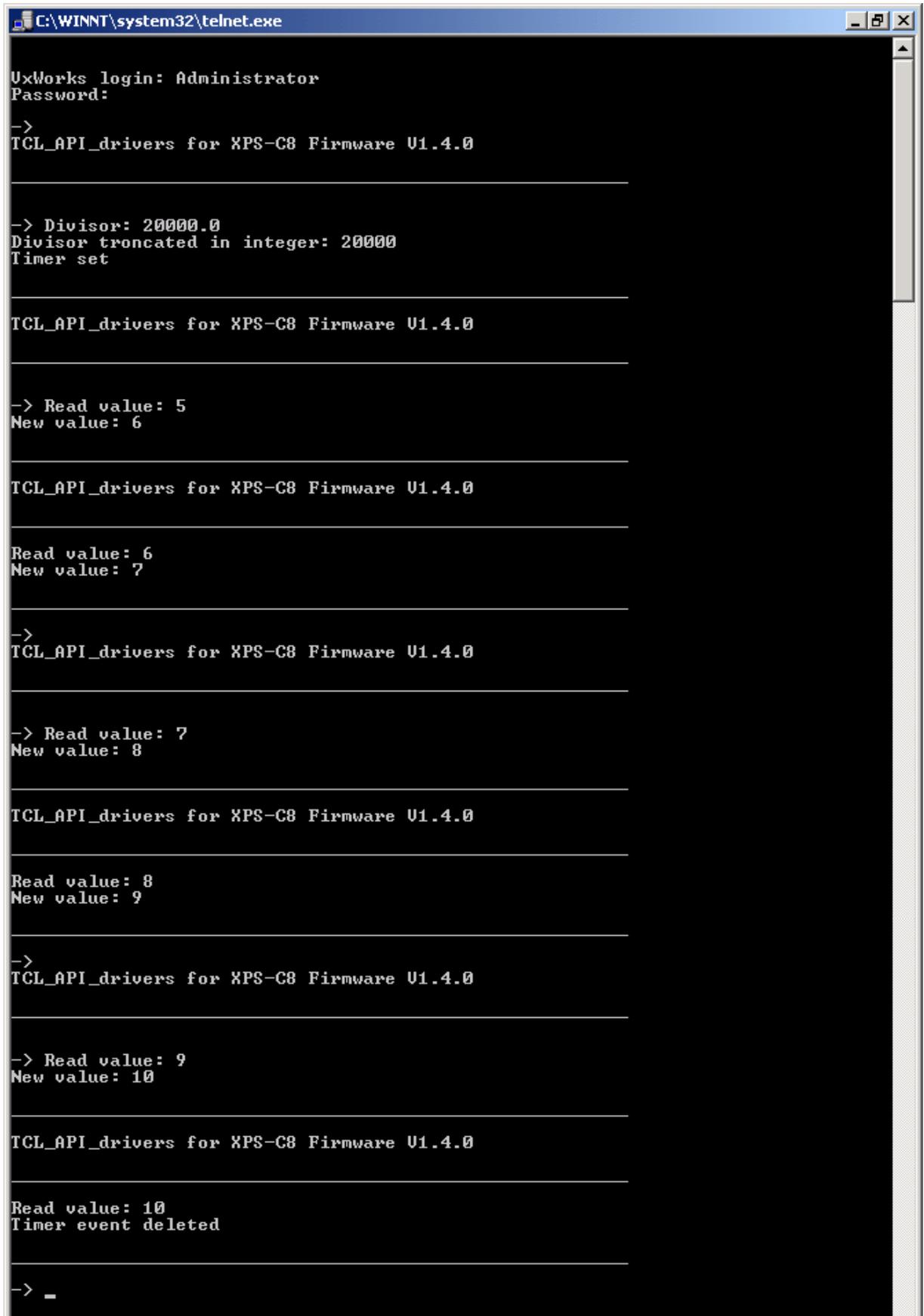
# Initialization
set TCPTimeOut 0.5
set code 0
set GlobalVarNumber 1
set ReadValue 0
set NewValue 0
set END 10
set Positioner "SINGLE_AXIS.MY_STAGE"
set EventName "Timer1"
set EventPara 0
# Open TCP socket
set code [catch "OpenConnection $TCPTimeOut socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Read global variable
set code [catch "GlobalArrayGet $socketID $GlobalVarNumber
ReadValue"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GlobalArrayGet"
    return
} else {
    puts stdout "Read value: $ReadValue"
}

if {$ReadValue < $END} {
# Increment global variable
set NewValue [expr {$ReadValue + 1}]
# Set global variable to a new value
set code [catch "GlobalArraySet $socketID $GlobalVarNumber
$NewValue"]
if {$code != 0} {

```

```
        DisplayErrorAndClose $socketID $code "GlobalArraySet"
        return
    } else {
        puts stdout "New value: $NewValue"
    }
} else {
    # Delete timer event
    set code [catch "EventRemove $socketID $Positioner
$EventName $EventPara"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "EventRemove"
        return
    } else {
        puts "Timer event deleted"
    }
}
}
# close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
}
```

This is what gets displayed on a Telnet window for the above example. For details about Telnet connections, see section 5 Telnet connection:



```
C:\WINNT\system32\telnet.exe

UxWorks login: Administrator
Password:
->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> Divisor: 20000.0
Divisor troncated in integer: 20000
Timer set

-----

TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> Read value: 5
New value: 6

-----

TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

Read value: 6
New value: 7

-----

->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> Read value: 7
New value: 8

-----

TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

Read value: 8
New value: 9

-----

->
TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

-> Read value: 9
New value: 10

-----

TCL_API_drivers for XPS-C8 Firmware U1.4.0

-----

Read value: 10
Timer event deleted

-----

-> _
```

## 7.13 TCL script with input arguments

### Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

### Description

This example opens a TCP connection, kills the single axis group, then initializes and homes it. It reads the three required input arguments: the start position, the end position, and the number of cycles for moving from the target position to the end position. When the user enters via the web site interface the arguments 10, -10 and 3, the positioner moves from -10 to +10 three times. Then, the program ends by closing the socket.

Please see the chapter 6 *Error handling* for the code of the procedure *DisplayErrorAndClose*.

### Code

```
# Initialization
set Timeout 20
set code 0
set group "SINGLE_AXIS"
# Open TCP socket
set code [catch "OpenConnection $Timeout socketID"]
if {$code != 0} {
    puts stdout "OpenConnection failed => $code"
} else {
# Recover the input arguments entered by the user
    if {$tcl_argc == 3} {
        set startpos $tcl argv(0)
        set endpos $tcl argv(1)
        set cycles $tcl argv(2)
    } else {
        puts stdout "Wrong number of parameters, 3 are needed"
        set code [catch "TCP CloseSocket $socketID"]
        return
    }
# Kill group
    set code [catch "GroupKill $socketID $group"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupKill"
        return
    }
# Initialize group
    set code [catch "GroupInitialize $socketID $group"]
    if {$code != 0} {
```

```
    DisplayErrorAndClose $socketID $code "GroupInitialize"
    return
}
# Home group
set code [catch "GroupHomeSearch $socketID $group"]
if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GroupHomeSearch"
    return
}
# Loop until the number of cycles (third parameter entered by the
user) is reached
for { set i 0 } {($i < $cycles) } {incr i} {
# Move group to start position
    set code [catch "GroupMoveAbsolute $socketID $group
$startpos"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupMoveAbsolute"
        return
    }
# Move group to end position
    set code [catch "GroupMoveAbsolute $socketID $group $endpos"]
    if {$code != 0} {
        DisplayErrorAndClose $socketID $code "GroupMoveAbsolute"
        return
    }
}
}
# Close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
```





Visit Newport Online at:  
[www.newport.com](http://www.newport.com)

**North America & Asia**

Newport Corporation  
1791 Deere Ave.  
Irvine, CA 92606, USA

**Sales**

Tel.: (800) 222-6440  
e-mail: [sales@newport.com](mailto:sales@newport.com)

**Technical Support**

Tel.: (800) 222-6440  
e-mail: [tech@newport.com](mailto:tech@newport.com)

**Service, RMAs & Returns**

Tel.: (800) 222-6440  
e-mail: [service@newport.com](mailto:service@newport.com)

**Europe**

MICRO-CONTROLE Spectra-Physics S.A.S  
9, rue du Bois Sauvage  
91055 Évry CEDEX  
France

**Sales**

Tel.: +33 (0)1.60.91.68.68  
e-mail: [france@newport.com](mailto:france@newport.com)

**Technical Support**

e-mail: [tech\\_europe@newport.com](mailto:tech_europe@newport.com)

**Service & Returns**

Tel.: +33 (0)2.38.40.51.55

