PowerTerm®
Power Script Language

Programmer's Reference
Version 6.5.1

Ericom North America
Ericom Software Inc.
231 Herbert Ave., Bldg. #4
Closter, NJ 07624 USA
Tel: +1 201 767 2210
Fax: +1 201 767 2205
Toll Free: 1 888 769 7876
Email: info@ericom.com

Ericom Europe
Ericom Software (UK) Ltd.
11a Victoria Square
Droitwich, Worcestershire
WR9 8DE United Kingdom
Tel: +44 (0)1905 777 970
Fax: + 44 (0)1905 777 972
Email: ukinfo@ericom.com

Ericom France
Ericom Software France
19, Boulevard Malesherbes
75008 Paris
France
Tel: +33 (0)1 5527 3938
Fax: +33 (0)2 4773 8765
Email: frinfo@ericom.com

Ericom International
Ericom Software Ltd.
8 Hamarpeh Street
Har Hotzvim
Jerusalem 91450 Israel
Tel: +972 (0)2 571 4774
Fax: +972 (0)2 571 4737
Email: info@ericom.com

Important Notice

This reference is subject to the following conditions and restrictions:

- This Programmer's Reference provides documentation for the PowerTerm Series of products. Your specific PowerTerm product might include only a portion of the features documented in this reference.
- The proprietary information belonging to Ericom® Software Ltd. is supplied solely for the purpose of assisting explicitly and properly authorized users of PowerTerm®.
- No part of its contents may be used for any other purpose, disclosed to any person or firm, or reproduced by any means, electronic and mechanical, without the express prior written permission of Ericom® Software Ltd.
- The text and graphics are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- The software describe in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of that agreement.
- Information in this document is subject to change without notice. Corporate and individual names and data used in examples herein are fictitious unless otherwise noted.

Table of Contents

Introduction to PSL

The **Power Script Language (PSL)** is PowerTerm's own programming language. It enables you to create scripts for automating tasks. For example, you can create a script to login to PowerTerm, execute a file, display a message, etc. Scripts can be run upon startup or during a PowerTerm session. They can be written in any standard text editor, like Notepad, and are saved with a **.psl** extension. PSL is intended for users with programming or scripting skils.

Each PSL command is also described in PowerTerm's online help, which can be accessed by selecting **Contents** from the **Help** menu.

The PSL Reference is comprised of the following topics:

- **PSL Overview,** describes the programming conventions used in PSL, page 10.
- **Using PowerTerm Scripts**, describes how to create, edit, run, save and activate scripts in PowerTerm, page 15.
- **PSL Reference**, describes standard scripts and commands as well as an alphabetical reference of all PSL commands with examples, page 19.

PSL Overview

The following topics are described:

- **PSL Types**, describes the different categories of PSL commands, below.
- **PSL Syntax**, describes the syntax used to create PowerTerm scripts, below.
- **PSL Data Types**, describes common formats for data strings, page 12.
- **Variable Assignment**, describes the syntax used to assign variables, page 13.

PSL Types

The PSL commands can be grouped into different categories:

Simulation transmission to host commands	Enables you to communicate with the host. For examples, the **<send>** command sends data to the host.
Standard programming commands	Enables you to use standard programming commands. For example, the **<exec>** command opens a program.
File handling commands	Enables you to work with files. For example, the **<read>** command reads from a file.
PowerTerm-specific commands	Enables you to activate specific PowerTerm features. For example, the **<map>** command enables you to map a PC key to a host key.
Desktop interface commands	Enables you to manipulate components in the PowerTerm window. For example, the **<menu hide>** command hides the PowerTerm menu.
DDE commands	Enables you to use standard Microsoft Windows DDE mechanisms to communicate with other Windows applications.

PSL Syntax

A command consists of one or more fields separated by spaces or tabs. The first field is the name of a command, which may be either a built-in command or a procedure consisting of a sequence of PSL commands. Newline characters are used as command separators, and semicolons may be used to separate commands on the same line. Each PSL command returns either a string result, or an empty string. PSL commands must be entered in lower case.

PSL has four additional syntactic constructions:

- **Braces {}**
- **Brackets []**
- **Dollar sign $**
- **Backslash **

Braces

Braces are used to group complex arguments. They act as nestable quote characters. If the first character of an argument is an open brace, then the argument is not terminated by white space. Instead, it is terminated by the matching close brace. The argument passed to the command consists of everything between the braces, with the enclosing braces stripped off. For example:

```
host={vms unix{aix hp sun} aos}
```

The variable host will receive one argument:
"vms unix {aix hp sun} aos"
This particular command will set the variable host to the specified string.

If an argument is enclosed in braces, then none of the other substitutions described below is made on the argument. One of the most common uses of braces is to specify a PSL sub-program as an argument to a PSL command.

Brackets

Brackets are used to invoke command substitution. If an open bracket appears in an argument, then everything from the open bracket up to the matching close bracket is treated as a command and executed recursively by PSL. The result of the command is hten substituted into the argument in place of the bracketed string. For example:

```
msg=[format {Data is %s bytes long} 99]
```

The format command does print-like formatting (from the C language) and returns the string "Data is 99 bytes long", which is then assigned to the variable message.

Dollar Sign

The dollar sign is used for variable substitution. If the dollar sign appears in an argument, then the following characters are treated as a variable name, and the contents of the variable are substituted into the argument in place of the dollar sign and name. For example:

```
num=99
msg=[format {Data is %s bytes long} $num]
```

The result is the same as the single command in the previous example.

The following are examples of common functions for the dollar sign in PSL:

- **$P1** represents the variable of the parameter if P1 is the parameter.
- **$PC** counts the number of parameters in the command line.

Backslash

The backslash character may be used to insert special characters into arguments, such as braces or nonprinting characters. Such special characters include the following:

```
[ ]  { }  $  t  b  r  m
```

`\xFF` will send the hex code FF (where F can be any hex character: 0-9, A-F). For example:

```
send \x1B\[m
```

This command will send the three characters: escape, [, and m.

PSL Data Types

PSL uses only one type of data: **strings**. All commands, arguments to commands, results returned by commands and variable values are ASCII strings.
Although everything in PSL is a string, many commands expect their string arguments to have particular formats. There are three particularly common formats for strings:

- **Lists**
- **Expressions**
- **Commands**

Lists

A list is just a string containing one or more fields separated by white space, similar to a command. Braces may be used to enclose complex list elements. These complex list elements are often lists in their own right. For example:

```
{vms unix {aix hp sun} aos}
```

This is a list with four elements, the third of which is a list with three elements.

PSL provides commands for a number of list-manipulation operations, such as creating lists, extracting elements and computing list lengths.

Expressions

The second common form for a string is a numeric expression. PSL expressions have the same operators and rules of precedence as expressions in the C language. The **expr** PSL command evaluates a string as an expression and returns the result (as a string, of course). For example:

```
Expr {($x<$y)||($z!=0)}
```

Returns “1” if the numeric value of variable *x* is less than that of variable *y*, or if variable *z* is not zero. Otherwise it returns “0”. Several other commands, such as **if** and **for**, expect one or more of their arguments to be expressions.

Commands

The third common interpretation of strings is as commands (or sequences of commands). Arguments of this form are used in PSL commands that implement control structures. For example:

```
if{$x<$y}{
swap=$x
x=$y
y=$swap
}
```

The **if** command receives two arguments here, each of which is delimited by braces. **if** is a built-in command that evaluates its first argument as an expression. It executes its second argument as a PSL command, if the result is non-zero. This particular command swaps the values of the variables *x* and *y* if *x* is less than *y*.

PSL also allows users to define command procedures written in the PSL language. The **proc** built-in command is used to create a PSL procedure (**PSLproc**). For example:

```
proc factorial x{
if{$x==1}{return 1}
return[expr{$x*[factorial[expr$x-1]]}]
}
```

This PSL command defines a recursive factorial procedure. The **proc** command takes three arguments: a name of the new PSLproc, a list of variable names (in this case the list has only a single element, *x*), and a PSL command that comprises the body of the PSLproc. After this proc command has been executed, *factorial* may be invoked just like any other PSL command for example:

```
Factorial 4
```

Returns the string “24”.

In addition to the commands already mentioned, PSL provides commands for manipulating strings (comparison, matching and printf/scanf-like C language operations), commands for manipulating files and file names. The built-in PSL commands provide a simple but complete programming language.

Variable Assignment

Syntax

```
VarName = value
varName[index] = value
```

Variable assignment is as follows: the variable *varName* is on the left side of the expression and the value you want to assign to the variable is on the right. For example:

```
B = 200
```

Variables

There are two types of variables:

- **Scalar**
- **Array**

A variable containing a single value is a **scalar** variable and the majority of the time fits ones needs. Other times, it's convenient to assign more than one related value to a single variable. Then you can create an **array** variable that can contain a series of values. Scalar and array variables are declared in the same way, except that the declaration of an array variable uses brackets [] following the variable name.

A variable's scope is determined by where it is declared. When you declare a variable within a procedure, only code within that procedure can access or modify the value of that variable. It has local scope and is called a **local variable**. In contrast a **global variable** declared outside a procedure is recognizable to all the procedures in your script.

N*OTE An exception to the rule governing local variables is when the global command has been invoked to declare* varName *to be global.*

Using PowerTerm Scripts

PowerTerm provides you with the following script options:

- **Create a Script** creates a script to run upon startup or at any time during a PowerTerm session.
- **Edit a Script** edits an existing script file.
- **Record a Script** creates a script by recording all the actions that you perform in the PowerTerm window. Actions can include selecting a menu option, typing an entry on the screen, making selections in a dialog box and so on.
- **Run Scripts** runs specific scripts or individual commands, upon startup or during a PowerTerm session, to automate specific tasks. You can only run saved scripts.
- **Activate a recorded Script** executes a non-saved script from the current memory.
- **Save a recorded Script** saves your scripts to be used at a later date.

➢ To create a script file:

1. Select **Script | Edit Script**. The **Edit Script** dialog box appears.
2. Specify the PowerTerm folder in the **Look in** field in which to store the new script.
3. Type a name for the new script file in the **File name** field. You can type any name and extension that comply with DOS file-naming conventions. It is recommended that you use a **.psl** extension, because PowerTerm automatically recognizes it indicating a script file.
4. Click **Open**. A message box displays, asking whether to create a new file.
5. Click **Yes**. Notepad opens.
6. Type the script command(s) that you require.
7. Select **File | Save**.
8. Exit from Notepad.

➢ To edit a script file:

1. Select **Script | Edit** Script. The **Edit Script** dialog box appears.
2. Double-click the required script file in the files list. Notepad opens and displays the selected script.
3. Edit the script as required.
4. Select **File | Save**.
5. Exit from Notepad.

➢ To record a script:

1. Select **Script | Start Script Recording**. The menu option changes to **Stop Script Recording**.
2. Perform the manual operations that you want to record. For example, select a menu option, enter parameters in a dialog box, or type a password.

3. Select **Script | Pause Script Recording** if you do not want to record certain operations. The script recording process pauses and the menu option changes to **Continue Script Recording**.
4. Select **Script | Continue Script Recording** to resume script recording.
5. Select **Script | Stop Script Recording** when you have performed all the operations to be stored in the script.

➢ To activate a recorded script:

- Select **Script | Activate Recorded Script**. The script currently recorded in memory is activated.

➢ To save a recorded script:

1. Select **Script | Save Recorded Script**. The **Record Script** dialog box appears.
2. Select the directory in which you want to save the file.
3. Enter a file name. The .psl file extension is automatically added.
4. Click **Save**. The file is saved with the specific file name.

Running Scripts

PowerTerm enables you to run scripts from startup by creating a shortcut to PowerTerm and a specific script file. This option can be used to connect to each host using different scripts. You can also run scripts by programming Soft buttons and the Power Pad (Windows edition).

➢ To run individual script commands:

1. Select **Script | Script Command**. The **Script Command** dialog box appears.
2. Type the name of the script command you want to run.
3. Click **OK**. The specified script command is executed.

➢ To run a script file:

1. Select **Script | Run Script**. The **Run Script** dialog box appears with a list of all the files in the PowerTerm directory that carry the .psl extension.
2. Double-click the script file that you want to run. The selected script file is executed.

➢ To run a script file upon connecting to a host:

1. Select **Communication | Connect**. The **Connect** dialog box appears.
2. Type the desired script file in the **Script File** field or browse for it.

➢ To start PowerTerm using a script (Windows edition):

The following procedure describes one way to create a shortcut. Consult your Windows documentation for a description of other available options.

1. Locate the file **ptw23.exe** on your computer.
2. Right-click and select **Create Shortcut** option. The **Shortcut to ptw32.exe** appears in the current folder.
3. Right-click the created shortcut and select **Properties** option. The **Shortcut to ptw32.exe Properties** dialog box appears.

4 In the **Target** field, position your cursor after the **.exe** file name.

5 Add a space and then type the name of the required script file.
You can also add parameters to the script file. These determine the communication parameters. For example, it can be the name of the host to which you want to connect, or the port number.

1. Position your cursor after the PSL script name in the **Target** field
2. Add a space and type the required parameters. Parameters should be separated by a space. For example:

   ```
   \PTW32\PTW32.EXE COMM.PSL 1 9600 xonxoff
   ```

   PowerTerm recognizes Windows file naming conventions, including spaces in a file name. If you have a setup file with a space in the name, PowerTerm ignores the space and looks directly for the .psl extension.

6 Click **OK**. When you start PowerTerm, the script file is automatically executed and you are connected to the host that you specified in your setup file.

Programming Soft Buttons

Along the bottom of the PowerTerm window are twelve programmable Soft buttons, by default named from **F1** to **F12**. These can be renamed and programmed to execute customized scripts. You send the programmed command to the host by clicking the desired Soft button.

The Soft button parameters are saved automatically in the terminal setup file.

➢ To program Soft buttons:

1 Right-click the Soft button that you want to program. The **Function Button** dialog box appears.

2 Type the **Function Description** (that is, the name that will appear on the button).

3 Type the **Script Commands** to be run by this button. For example, “exec notepad”. You can type several scripts separated by semicolons.

4 Click **OK**. The Soft button is now displayed with its new name and will execute the defined script.

➢ To run a script file during a PowerTerm session using Soft buttons:

- Click the Soft button that has the desired script assigned. The script is sent to the host.

Programming the Power Pad

The Power Pad is a floating keypad for which its buttons can be programmed to execute customized PSL scripts. The buttons are by default named **F1**, **F2**, **F3**, and so on, with a few default function names, such as **Clear**, **Enter** and **Insert**. The number of displayed buttons and their names can be changed. You send the programmed command to the host by clicking the desired Power Pad button.

➢ To program the Power Pad:

1 Select **Options | Show Power Pad**. The **Power Pad** appears.

2 Right-click the Power Pad button that you want to program. The **Power Pad Button** dialog box appears.

3 Type the **Button Description** (that is, the name that will appear on the Power Pad button).

4 Type the **Script Commands** to be run by this Power Pad button. For example:

```
send <f13>
```

You can type several scripts separated by semicolons.

5 Click **OK**. The Power Pad button is now displayed with its new name and will execute the defined script.

PSL Reference

The following topics are described:

- **PowerTerm Sample Scripts**, describes the standard scripts used in PowerTerm, below.
- **Activating Script Files from the Host**, describes commands to activate a script file or script commands via special escape sequences, below.
- **DDE Commands**, describes the function of DDE commands for both client and server application (Windows edition), page 20.
- **Alphabetical Reference**, describes the syntax and provides examples for each PSL command.

PowerTerm Sample Scripts

PowerTerm provides several sample scripts designed for frequent tasks. The following table lists part of the sample scripts and their parameters. Additional sample scripts are also included as part of PowerTerm.

Script	Parameters	Parameter Values
COMM.PSL	Port number	1 – 32
	Baud rate	All
	Protocol type	none, xonxoff, hardware
TELNET.PSL	Host name	Specify the name of the host.
LAT.PSL	Service name	Specify the name of the service.
CTERM.PSL	Node name	Specify the name of the CTERM node.

Activating Script Files from the Host

A host application may activate a script file or script commands via special escape sequences.

Escape Sequences for VT

Activating a script file called Script-Name:

```
ESCP$sScript-NameESC\
```

An example activating the *message.psl* script:

```
ESCP$smessage.pslESC\
```

Activating script commands called Script-Commands:

```
ESCP$tScript-CommandsESC\
```

An example activating the "*message testing; send end*" commands:

```
ESCP$tmessage testing ; send endESC\
```

NOTE ESC is the ASCII 27 code.

Escape Sequences for DG

Activating a script file called Script-Name:

```
ESCWsScript-Name000
```

Activating script commands called Script-Commands:

```
ESCWtScript-Commands000
```

NOTE ESC is the ASCII 30 code, 000 is the ASCII 0 code.

DDE Commands (Windows edition)

PowerTerm enables you to use the standard Microsoft DDE mechanism to communicate with other Windows applications. PowerTerm can be a DDE client application or a DDE server application. The DDE server application waits for requests from DDE clients, and allows them to supply it with information or receive information. For example if a spreadsheet DDE server will let clients get data from cells and put data into cells of a file.

As a DDE server, PowerTerm uses the server name **ptw** with topic **psl**. Any application can request it to execute commands and return the related return data.
A client application can access PowerTerm with the **dde execute** command or the **dde request** command and an item that is any valid PSL command separated with semicolon (;). The single DDE server PSL command is:

```
dde return value
```

After a DDE request command is executed, the PowerTerm DDE server sends the value from the last DDE return command executed in this request. If no DDE return command was executed, it returns an empty answer. Examples for PowerTerm as a DDE server might be:

- Sending information to the host.
- Reading information from the emulation screen.

As a DDE client, PowerTerm performs one of several DDE operations, depending on the option. The legal options are:

dde initiate application Name topicName	Connects to the *applicationName* DDE server with *topicName*. Returns a conversation ID for use with successive DDE commands.
dde execute convId command	Executes a server command. Returns an empty string.

dde request conId item	Returns an item from the server.
dde poke convId item value	Changes an item of the server to the new value. Returns an empty string.
dde terminate convId	Terminates a DDE conversation with the server.
dde returns	Returns a value according to the option.

Example 1:

Assigns three numbers on the emulation screen to array "cel":

```
for {i=1}{$i<4}{incr i}{
      row=[expr $i+3]
      cel($i)=[screen-rect $row 10 $row 15]}
```

Initiates a DDE conversation with Microsoft Excel file TEST.XLS:

```
conv=[dde initiate EXCEL TEST.XLS]
```

Pokes the three numbers to three cels in TEST.XLS:

```
for{i=1}{$i<4}{incr i}{dde poke $conv R1C$I $cel ($i)}
```

Requests the sum of those numbers from a result cel in TEST.XLS:

```
sum=[dde request $conv "R2C1"]
```

Terminates the DDE conversation:

```
dde terminate $conv
```

Sends the result to the host application:

```
send $sum
```

Example 2:

Initiates a DDE conversation with another PowerTerm, which is connected to another computer. Reads information from the screen (of the other host) and sends it to its own host:

```
conv=[dde initiate PTW PSL-B]
data=[dde request $conv{
      dde return[screen 10 1 15 80]}]
dde execute $conv{send joe}
```

For more explanations of the DDE commands, see the alphabetical commands reference on following pages.

A

activate-menu-item

Description
Activates a PowerTerm menu item.

Syntax
activate-menu-item *itemname*

itemname Specifies the name of the PowerTerm menu item to be activated.

Notes
If the name of the menu item is more than one word, it should be specified without any spaces. See example below.
This command activates the specified menu item even if the menu item was removed.

Returns
N/A

Example
To activate the PowerTerm Communication menu item labeled “Send File”:

```
activate-menu-item SendFile
```

To activate the PowerTerm File menu item labeled “Save Terminal Setup As”:

```
activate-menu-item SaveTerminalSetupAs
```

append

Description
Appends to a variable

Syntax
append *varName value* [*value value ...*]

varName Specifies name of a variable.

Notes

Appends all of the value arguments to the current value of variable varName. If varName does not 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.

Returns

N/A

Example

The following line appends variables y and z to variable x:

```
append x $y $z
```

It is much more efficient, if $x is long, than:

```
x = $x$y$z
```

array

Description

Manipulates array variables.

Syntax

array *option arrayName* [*arg arg ...*]

option Specifies a valid option.
ArrayName Specifies the name of an existing array variable.

Notes

This command performs one of several operations on the variable given by *arrayName*. *ArrayName* must be the name of an existing array variable. The option argument determines what action is carried out by the command. A description of each valid option (which may be abbreviated) follows:

array *anymore arrayName searched*

- 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, because the return value from array *nextelement* does not indicate whether the search has been completed.

array *donesearch arrayName searched*

- This command terminates an array search and destroys all the states associated with that search.
- *searchId* indicates which search on *arrayName* to destroy, and must have been the return value from a previous

invocation of array *startsearch*.
- Returns an empty string.

array *names arrayName*
- Returns a list containing the names of all of the elements in the array. If there are no elements in the array, then an empty string is returned.

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

array *size arrayName*
- Returns a decimal string giving the number of elements in the array.

array *startsearch arrayName*
- This command initializes an element-by-element search through the array given by *arrayName*, such that invocations of the array *nextelement* command will return the names of the individual elements in the array.
- The array *donesearch* command should be invoked, when the search has been completed.
- 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.

Returns
N/A

ascii-code

Description
Returns the ASCII code for the specified character.

Syntax
ascii-code *character*

character Specifies the character for which its ASCII code is returned.

Returns
N/A

Example

The following example will display the ASCII value for the character, which has the ASCII value of 97:

```
get_ascii =[ascii-code a]
message $get_ascii
```

B

break

Description
Aborts a looping command.

Syntax
break

Notes
This command may be invoked only inside the body of a looping command such as **for**, **foreach**, or **while**.

Returns
N/A

Example
break terminates the **while** loop when variable *x* equals *5*:

```
x = 0
while {$x < 10}{
      incr x
      if {$x == 5}
            break
      .
      .
      .
}
```

case

IMPORTANT The case command has been deprecated and is supported only for backward compatibility. At some point in the future it may be removed entirely. You should use the switch *command instead.*

Description

Evaluates one of several scripts, depending on a given value.

Syntax

case *string* in *patList body patList body ...*

string Compares to each of the *patList* arguments to find a match in the order that they appear.
patList Consists of a single pattern or list of patterns.

Notes

The **case** command matches *string* against each of the *patList* arguments in order. Each *patList* argument is a list of one or more patterns. If any of these patterns matches *string* then **case** evaluates the following *body* argument by passing it recursively to the interpreter and returns the result of that evaluation. Each *patList* argument consists of a single pattern or list of patterns. Each pattern may contain any of the wild cards described under **string match**. If a *patList* argument is **default**, the corresponding body will be evaluated if no *patList* matches *string*. If no *patList* argument matches *string* and no default is given, then the **case** command returns an empty string.
Two syntaxes are provided for the *patList* and *body* arguments. The first uses a separate argument for each of the patterns and commands. This form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument. The argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line case commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *patList* arguments are in braces in the second form, no command or variable substitutions are performed on them. This makes the behavior of the second form different than the first form in some cases.

Returns

N/A

Example

Returns ‘3’:

```
case abc in {a b} \
      {format 1} default {format 2} a* {format 3}
```

Returns '1':

```
.ta .5c 1c
case a in {
      {a b} {format 1}
      default {format 2}
      a* {format 3}
}
```

Returns '2':

```
case xyz {
      {a b}
            {format 1}
      default
            {format 2}
      a*
            {format 3}
}
```

catch

Description

Evaluates script and traps exceptional returns.

Syntax

catch *script varName*

script	Specifies the script that is run and scrutinized for errors.
varName	Specifies the error message from interpreting *script*.

Notes

The **catch** command may be used to prevent errors from aborting command execution. **catch** calls the interpreter recursively to execute *script*, and always returns without raising an error, regardless of any errors that might occur while executing *script*.

If *script* raises an error, **catch** will return a non-zero integer value corresponding to one of the exceptional return codes. If the *varName* argument is given, then the variable it names, is set to the error message from interpreting *script*.

If *script* does not raise an error, **catch** will return 0 and set the variable to the value returned from *script*.

catch catches all exceptions, including those generated by **break** and **continue** as well as errors. The only errors that are not caught are syntax errors found when the script is compiled. This is because the **catch** command only catches errors during runtime. When the catch statement is compiled, the script is compiled as well and any syntax errors will generate an error.

Returns

Returns zero if there were no errors. Otherwise it will return a non-zero value corresponding to one of the exceptional return codes.

Examples

The **catch** command may be used in an **if** statement to branch based on the success of a script.

```
if {[catch {open $someFile w} fid]}{
    puts stderr "Could not open $someFile for writing\n$fid"
    exit 1
}
```

The **catch** command will not catch compiled syntax errors. The first time **proc foo** is called, the body will be compiled and an error will be generated.

```
proc foo {}{
    catch {expr {1 +- }}
}
```

cd

Description

Changes the working directory.

Syntax

cd *dirName*

dirName Specifies the name of the new current working directory.

Returns

Returns an empty string.

Example

Changes working directory to pterm:

```
cd pterm
```

clear screen

Description
Clears the screen.

Syntax
clear-screen

Returns
N/A

close

Description
Closes an open file.

Syntax
close *fileId*

fileId Specifies the file to be closed.

Notes
fileId must be the return value from a previous invocation of the **open** command. After this command, it should not be used anymore.

Returns
The normal result of this command is an empty string, but errors are returned if there are problems in closing the file.

Example
Opens file *"sales.dat"*, reads 10 bytes, closes it, and displays the data in a message box:

```
fileId = [open sales.dat]
data = [read $fileId 10]
close $fileId
message $data
```

color

Description

Sets the foreground and background colors on a color graphics monitor, or switches output to the color monitor.

Syntax

color *{attributes}{foreground}{background}*

Valid choices:

attributes
normal
dim
bold
blink
reverse
underline

background
black
magenta
dark-gray
dark-magenta
blue
cyan
dark-blue
dark-cyan
red
yellow
dark-red
dark-yellow
green
gray
dark-green
white

foreground
black
magenta
dark-gray
dark-magenta
blue
cyan
dark-blue
dark-cyan
red
yellow
dark-red
dark-yellow

green
gray
dark-green
white

Notes
On a system with both a color and monochrome display monitor, the **color** command switches output from the mono to the color display.
The choice of attributes depends on the particular terminal emulation and its supported screen attributes.

Returns
N/A

Example
To set the color to white characters on a blue background:

```
color normal white blue
```

concat

Description
Joins lists together.

Syntax
concat *arg* [*arg* ...]

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

Returns
Returns a single list with all elements.

Example
Returns {a b c d e f {g h}}:

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

continue

Description
Skips to the next iteration of a loop.

Syntax
continue

Notes
This command may be invoked only inside the body of a looping command such as **for**, **foreach**, or **while**.
It signals the innermost containing loop command to skip the remainder of the loop's body but to continue with the next iteration of the loop.

Example
Skips over the **while** loop commands when variable x is less than 5.

```
x = 0
while {$x < 10} {
      incr x
      if {$x < 5}
            continue
      .
      .
      .
}
```

copy file

Description
Copies a file.

Syntax
copy-file *existing-filename new-filename*

existing-filename	Specifies the name of the existing file.
new-filename	Specifies the name of the new file.

Notes
If the *new-filename* exists, it will be overwritten.

Returns
N/A

Example
Copies a file named *sales.dat* and gave it the name *Jerry_sales.dat*:

```
copy-file sales.dat Jerry_sales.dat
```

copy from clipboard

Description
Returns current string from clipboard.

Syntax
copy-from-clipboard

Returns
N/A

Example
One word copied to clipboard:

```
copy-to-clipboard
a = [copy-from-clipboard]
message $a
```

copy to clipboard

Description
Copies designated text to clipboard.

Syntax
copy-to-clipboard *text*

text Specifies the string that is being copied to the clipboard.

Notes
If the specified text is more than one word long then it must be enclosed in quotation marks (“ ”).

Returns
Returns the content of the clipboard in a string format.

Example
One word copied to clipboard:

```
copy-to-clipboard schedules
```

Multiple words copied to clipboard:

```
copy-to-clipboard “Tom and Harold’s schedules”
```

cursor

Description
Moves the cursor to a new position.

Syntax
cursor *row col*

row Specifies the row position to where the cursor is to be positioned.
col Specifies the column position to where the cursor is to be positioned.

Notes
The cursor command changes the cursor position but does not send any indication to the host.

Returns
Returns an empty string.

Example
```
cursor 4 34
```

date

Description
Obtains the current date from the system.

Syntax
date

Returns
N/A

Example
```
a = [date]
message $a
```

dde execute

Description
Executes a server command.

Syntax
dde execute *conv-id command-string*

conv-id	Specifies the returned value from the **dde initiate** command.
command-string	Contains the desired dde command to be executed.

Returns
Returns an empty string.

Example
```
conv = [dde execute EXCEL TEST.XLS]
```

dde initiate

Description
Connects to the *applicationName* DDE server with *topicName*.

Syntax
dde initiate *applicationName topicName*

applicationName	Specifies the application to which PowerTerm is connecting.
topicName	Specifies the field designator (in Excel) or a paragraph or item (in WORD).

Returns
Returns a conversation id for use with successive dde commands.

Example
Launches EXCEL and displays the PRODUCTS spreadsheet:

```
conv = [dde initiate EXCEL PRODUCTS.XLS]
```

dde poke

Description
Inserts designated value(s) into specified location in server program.

Syntax
dde poke *convId item value*

convId	Specifies the returned value from the **dde initiate** command.
Item	Specifies the string containing the item ID that a change in value is requested.
value	Specifies the string containing the desired value.

Returns
Returns an empty string.

Example
Pokes the three numbers to three cells in TEST.XLS.

```
for {i = 1}{$i < 4}{incr i}{dde poke $conv R1C$i $cel($i)}
```

dde server topic

Description
Returns an item from the server when a client attempts to connect to it with a given topic.

Syntax

dde server-topic *application topic script-name*

application	Specifies to which server to connect to using the DDE protocol.
topic	Specifies the string containing the desired topic.
script-name	Specifies the string, which contains the method that should run on the server.

Notes

The server name is usually the name of the program itself, without the .exe extension. For more information about the syntax of dde macros, and its defined topic names, see page 20 or consult Microsoft documentation on the topic.

Returns

If the server recognizes the topic, a method specializing on the server should return an instance of one of the server's topic classes.
If the server does not recognize the topic, the method should return an empty string.

Example

If an application implements a topic for each open file, the topics **foo, foo.doc** and **c:\foo.doc** may all be acceptable strings for referring to the same topic. However **dde-server-topics** should return each topic once only.

display

Description

Displays a string on the current cursor position.

Syntax

display *string*

string	Specifies the string that is being displayed.

Notes

The **display** command displays the string on the screen, but does not send it to the host.

Returns

Returns an empty string.

Example

```
display "Hit ENTER to continue"
```

E

eof

Description
Checks the end-of-file condition on an open file.

Syntax
eof *fileId*

fileId Specifies the name of requested file upon which the end-of-file condition is checked.

Return
Returns 1 if an end-of-file condition occurred on *fileId*, otherwise 0.
fileId must have been the return value from a previous call to open.

Example
Opens file *input.dat* for reading and file *output.dat* for writing. While not end-of-input file, reads a line and writes it to the output file. Closes both files.

```
inFile = [open input.dat]
outFile = [open output.dat w]
gets $inFile data
while {! [eof $inFile]} {
      puts $outFile $data
      gets $inFile data
}
close $inFile
close $outFile
```

error

Description

Generates an error.

Syntax

error *message* [*info*] [*code*]

message	Specifies the error message that is displayed.
info	Contains the information that is used to initialize the global variable **errorInfo**.
code	Contains machine-readable description of the error in cases where such information is available.

Notes

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 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 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. If the *code* argument is not present, then **errorCode** is automatically reset to *NONE* by the interpreter as part of processing the error generated by the command.

Returns

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

eval

Description

Evaluates a PSL script.

Syntax

eval *arg* [*arg* ...]

arg Specifies parameter to be used with the command.

Notes

eval takes one or more arguments, which together comprise a PSL script containing one or more commands.
eval concatenates all its arguments in the same fashion as the **concat** command and passes the concatenated string to the PowerTerm PSL engine recursively.

Returns
Returns the result of the evaluation or any error generated by it.

Example
Assigns command variable with the **expr** command, executes the command, and displays its output:

```
command = "expr 3 * 8"
result = [eval $command]
message $result
```

exec

Description
Invokes a program.

Syntax
exec *program*

program Specifies the program to be activated

Notes
This command executes a program. The *program* variable may contain parameters.

Returns
N/A

Example
Activates the Notepad program with parameter *pt.psl*:

```
exec "notepad pt.psl"
```

exit emulator

Description
Closes the PowerTerm emulation program.

Syntax
exit-emulator

Returns
N/A

exit script

Description
Interrupts the currently running PSL script.

Syntax
exit-script

Notes
A confirmation message is displayed.

Returns
N/A

Example
The following will launch WordPad and then exit the script by displaying a confirmation screen before the script displays a message:

```
exec wordpad
exit-script
message Completed!
```

expr

Description
Evaluates an expression.

Syntax
expr *arg* [*arg arg* ...]

arg Specifies which expression to be evaluated.

Notes
Concatenates *args* while adding separator spaces between them, evaluates the result as a PSL expression, and returns the value.
The operators permitted in PSL expressions are a subset of the operators permitted in the C language expressions, and they have the same meaning and precedence as the corresponding C language operators. Expressions almost always yield numeric results (integer or floating-point values). PSL expressions support non-numeric operands and string comparisons. For example, the command

```
expr 8.2 + 6
```

evaluates to 14.2.

A PSL expression consists of a combination of **operands**, **operators**, and **parentheses**. White space may be used between the operands and operators and parentheses. The expression processor ignores it.

Operands

Operands can be interpreted as:

integer values	Most common. May be specified in • **decimal** which is the normal case • **octal** if the first character of the operand is 0 • **hexadecimal** if the first two characters of the operand are *0x*
floating-point number	An operand is treated as a floating-point number, if possible, and when it does not have one of the integer formats. They may be specified in any of the ways accepted by an ANSI-compliant C language 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`
string	An operand is left as a **string** when no numeric execution is possible. Only a limited set of operators may be applied to it.

Operands may be specified in any of the following ways:

- As a **numeric value**, either integer or floating-point.
- As a **PSL variable**, using standard $ notation. The variable's value will be used as the operand.
- 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.
- 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.
- As a **PSL command enclosed in brackets**. The command will be executed and its result will be used as the operand.
- 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 (for example, inside quoted strings), they are performed by the expression processor. However, the command parser may already have performed an additional layer of substitution 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 example, the variable *x* has the value 3 and the variable *y* has the value 6. 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 + $x                  6.1

expr 2 + "$x.$y"               5.6

expr 4 * [llength "6 2"]       8

expr {word one} < "word $x"    0
```

Operators

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

- ~ **!**	Unary minus, bit-wise NOT, logical NOT. • None of these operands may be applied to string operands. • Bit-wise NOT may be applied only to integers.
* / **%**	Multiply, divide, remainder. • None of these operands may be applied to string operands. • Remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.
+ -	Add and subtract. • Valid for any numeric operands.
<< >>	Left and right shift. • Valid for integer operands only.
< > <= >=	Boolean less, greater, less than or equal, and greater than or equal. • Each operator produces 1 if the condition is true, 0 otherwise. • These operators may be applied to strings as well as numeric operands, in which case string comparison is used.
== **!=**	Boolean equal and not equal. • Each operator produces a zero/one re • 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 the C language.
- 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.

For more details on the results produced by each operator, see the C language manual.

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 {$z ? [x] : [y]}
```

only one of [x] or [y] will actually be evaluated, depending on the value of $z. However, this is only true if the entire expression is enclosed in braces. Otherwise, PSL will evaluate both [x] and [y] before invoking the **expr** command.

Mathematical functions

PSL supports the following mathematical functions in expressions:

- **acos**
- **asin**
- **atan**
- **atan2**
- **ceil**
- **cos**
- **cosh**
- **exp**
- **floor**
- **fmod**
- **hypot**
- **log**
- **log10**
- **pow**
- **sin**
- **sinh**
- **sqrt**
- **tan**
- **tanh**

Each of these functions invokes the C language math library function of the same name.

Conversion Functions

PSL also implements functions for conversion between integers and floating-point numbers. These functions are:

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

double(arg)	If *arg* is a floating value, returns *arg*. Otherwise, converts *arg* to floating and returns the converted value.
int(arg)	If *arg* is an integer value, returns *arg*. Otherwise, converts *arg* to integer by truncation and returns the converted value.
round(arg)	If *arg* is an integer value, returns *arg*. Otherwise, converts *arg* to integer by rounding and returns the converted value.

Types, Conversion and Precision

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.
For example, the following commands 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 back to the string "18".

```
expr "0x03" > "2"
expr "0y" < "0x12"
```

F

F

file

Description

Manipulates file names and attributes.

Syntax

file *option name* [*arg arg ...*]

option Indicates what to do with the file name. Any unique abbreviation for *option* is acceptable.

name Specifies the name of a file to which its name and attributes are to be manipulated.

Returns

This command provides several operations on a file's name or attributes. The valid options are:

file atime *name*

Returns a decimal string giving the time at which *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 does not exist or its access time cannot be queried, then an error is generated.

file dirname *name*

Returns all of the characters in *name* up to but not including the last slash character.

- If there are no slashes in *name*, then returns dot (.).
- If the last slash in *name* is its first character, then returns "\".

file exists *name*

Returns 1 if *name* exists, 0 otherwise.

file extension *name*

Returns all of the characters in *name* after and including the last dot (.).

- If there is no dot in *name*, then returns the empty string.

file isdirectory *name*

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

file isfile *name*

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

file mtime *name*

Returns a decimal string giving the time at which *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 does not exist or its modified time cannot be queried, then an error is generated.

file rootname *name*

Returns all of the characters in *name* up to but not including the last dot (.) character in the name.

- If *name* does not contain a dot, then returns *name*.

file size *name*

Returns a decimal string giving the size of *name* in bytes.

- If the file does not exist or its size cannot be queried, then an error is generated.

file stat *name varName*

Returns an empty string.

- Invokes the **stat** system call on *name*, and uses the variable given by *varName* to hold information returned from the system call.
- *varName* is treated as an array variable, and the following elements of that variable are set: *atime, ctime, dev, mode, mtime, size, type*.
- Each element except *type* is a decimal string with the value of the corresponding field from the **stat** return structure.
- The *type* element gives the type of the file in the same form returned by the command file type.

file tail *name*

Returns all of the characters in *name* after the last backslash.

- If *name* contains no backslashes, then returns *name*.

file type *name*

Returns a string giving the type of *name*, which will be either file or directory.

flush

Description

Flushes buffered output for a file.

Syntax

flush *file*

file Specifies the return value from a previous call to open.

Returns

Returns an empty string.

Example

Opens file *sales.dat* for writing. Writes 10 lines, flushes data to file, and then closes the file.

```
fileId = [open sales.dat w]
for {i = 0} {$i < 10} {incr i} {puts $fileId "Line no $i"}
flush $fileId
close $fileId
```

for

Description

Executes a **for** loop.

Syntax

for *start test next body*

start	Initializes the loop variable.
test	Specifies the condition that must be met for the loop to end.
next	Specifies the increment for the loop variable.
body	Specifies the PSL command to be executed.

Notes

for is a looping command, similar in structure to the C language for statement. The *start*, *next*, and *body* arguments must be PSL command strings, and *test* is an expression string.
The **for** command first invokes PSL to execute *start*. Then it repeatedly evaluates *test* as an expression.

- If the result is non-zero it invokes PSL on *body*, then invokes PSL 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 PSL 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 the C language.

Returns

Returns an empty string.

Example

Executes the message command 10 times:

```
for {i = 1}{$i <= 10}{incr i}{message "i = $i"}
```

foreach

Description
Iterates over all elements in a list.

Syntax
foreach *varname list body*

varname	Specifies the name of a variable.
list	Specifies a list of values to assign to *varname*.
body	Specifies the PSL command to be executed.

Notes
For each element of *list* (in order from left to right), **foreach** assigns the contents of the field to *varname* as if the index command had been used to extract the field, then calls the PowerTerm PSL engine to execute *body*. The **break** and **continue** statements may be invoked inside *body*, with the same effect as in the **for** command.

Returns
Returns an empty string.

Example
For each item in the list, a message is displayed:

```
foreach var {Item1 Item2 Item3}{message "Item : $var"}
```

format

Description
Formats a string in the style of *sprintf*.

Syntax
format *formatString* [*arg arg ...*]

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.

Notes
This command generates a formatted string in the same way as the ANSI C *sprintf* procedure.

Returns and Formatting
The return value is the formatted string.

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 "%", 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 articular 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 have another five fields:

- A set of flags
- A minimum field width
- A precision
- A length modifier
- 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.
- If there are any positional specifiers in *formatString*, then all of the specifiers must be positional.

The available **flag character** options are:

\-	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 is not a sign.
0	Specifies that the number should be padded on the left with zeros 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 zeros should not be removed.

The **minimum field width** is determined by a number. 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 zeros 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 **precision** portion of a conversion specifier consists of a period followed by a number. The number is used in different ways for different conversions:

e , E, and f conversions	Specifies the number of digits to appear to the right of the decimal point.
g and G conversions	Specifies the total number of digits to appear, including those on both sides of the decimal point. However, trailing zeros after the decimal point will still be omitted unless the # flag has been specified.
integer conversions	Specifies a minimum number of digits to print. Leading zeros will be added if necessary.
s conversions	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 **length modifier** is one of the following:

h	Specifies that the numeric value should be truncated to a 16-bit value before converting. This option is rarely useful.
l	The modifier is ignored.

The **alphabetic character** 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 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 zeros 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.

func

Description

Creates, labels and provides the script for designated function buttons.

Syntax

func *buttonlabel action*

buttonlabel	Specifies the label on the function button.
action	Specifies the script to be performed when the function button is pressed.

Returns

N/A

Example

This example will create 2 functions buttons:

```
func 1 Greeting1 "send Hello"
func 2 Greeting2 "send goodbye"
```

G

get

Description

Returns an integer/string from a PowerTerm object.

Returns

get col

Returns the column number where the cursor is located in the emulation screen.

```
x = [get col]
message $x
```

get row

Returns the row number where the cursor is located in the emulation screen.

```
x = [get row]
message $x
```

get session-count

Returns the number of opened sessions in the current executable.

```
x = [get session-count]
message $x
```

get caps-lock

Returns 0 (off) or 1 (on).

```
x = [get caps-lock]
message $x
```

get answerback-message

Returns the answerback string message.

```
x = [get answerback-message]
message $x
```

get modem-name

Returns a string containing the name of the modem currently in use.

```
x  =  [get modem-name]
message $x
```

get modem-init-string Returns the initialization string for the modem currently in use.

```
x  =  [get modem-init-string]
message $x
```

get printer-name Returns string containing the name of the current default printer.

```
x  =  [get printer-name]
message $x
```

getenv

Description

Retrieves the value of an environment variable.
getenv searches the environment list for a string of the form name=value, and returns a pointer to the value in the current environment if such a string is present, otherwise a NULL pointer.

Syntax

getenv *environment-variable*

environment-variable Specifies the string, which is the object of the search in the current environment list.

Returns

Returns a pointer to static data, which can be overwritten by subsequent calls.

get printer name

Description

Retrieves the default printer name.

Syntax

get-printer-name

Returns

Returns the string representing the default printer.

gets

Description
Retrieves a file.

Syntax
gets *fileId* [*varName*]

fileId Specifies the name of requested file.
varName Specifies the variable name to which the string (line from the file) will be inserted.

Notes
This command reads the next line from the file given by *file* and discards the terminating newline character.

- If *varName* is specified, then the line is placed in the variable by that name and the return value is a count of the number of characters read, not including newline.
- If the end of the file is reached before reading any characters, then (-1) is returned and *varName* is set to an empty string.
- If *varName* is not specified, then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so **eof** may have to be used to determine what really happened.
- If the last character in the file is not a newline character, then **gets** behaves as if there were an additional newline character at the end of the file.

file must be the return value from a previous call to **open.** It must refer to a file that was opened for reading.

Returns
Returns the line length, if *varName* is specified, or the line itself.

Example
Gets first line from *sales.dat* file:

```
fileId = [open sales.dat]
gets $fileId data
close $fileId
```

glob

Description
Returns names of files that match patterns.

Syntax
glob *switches* [*pattern pattern ...*]

switches Specifies the option for the command.

Notes

This command performs file name "globbing" in a fashion similar to the CSH shell. 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 dot (.) 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.

Returns

Returns the files list of the files whose names match any of the pattern arguments.

Example

Displays all files with ".dat" extension:

```
message [glob *.dat]
```

global

Description

Accesses global variables.

Syntax

global *varname* [*varname* ...]

varname Specifies the names of the variables to be declared as global.

Notes

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

Returns

Returns an empty string.

Example

```
global varname1 varname2
```

H

history

Description
Manipulates the history list.

Syntax
history [*option*] [*arg*] [*arg ...*]

option Specifies in which particular manner the command operates.

Notes
The **history** command performs one of several operations related to recently executed commands recorded in a history list. Each of these recorded commands is referred to as an "event". When specifying an event to the **history** command, the following forms may be used:

- **A number** If positive, it refers to the event with that number (all events are numbered starting at 1). If the number is negative, it selects an event relative to the current event (**-1** refers to the previous event, **-2** to the one before that, and so on). Event **0** refers to the current event.
- **A string** Selects the most recent event that matches the string. An event is considered to match the string either if the string is the same as the first characters of the event, or if the string matches the event in the sense of the **string match** command.

Returns
The **history** command can take any of the following forms:

history — Same as **history info**, described below.

history add *command* **exec** — Adds the *command* argument to the history list as a new event.
- If **exec** is specified or abbreviated then the command is also executed and its result is returned.
- If **exec** isn't specified then an empty string is returned as result.

history change *newValue event* — Replaces the value recorded for an event with *newValue*. *event* specifies the event to replace, and defaults to the current event (not *event* **-1**).
- This command is intended for use in commands that

implement new forms of history substitution and wish to replace the current event, which invokes the substitution, with the command created through substitution.
- The return value is an empty string.

history event *event*

Returns the value of the event given by *event*.
- *event* defaults to **-1**.

history info *count*

Returns a formatted string, intended to be read, giving the event number and contents for each of the events in the history list except the current event.
- If *count* is specified then only the most recent *count* events are returned.

history keep *count*

This command may be used to change the size of the history list to count events. Initially, 20 events are retained in the history list.
- If *count* is not specified, the current keep limit is returned.

history nextid

Returns the number of the next event to be recorded in the history list. It is useful for things like printing the event number in command-line prompts.

history redo *event*

Re-executes the command indicated by *event,* and returns its result.
- *event* defaults to **-1**.
- This command results in history revision.

history substitute *oldhistorybuffer newhistorybuffer event*

Replaces the old history with the new one that which is indicated by *event* and returns its result.
- *event* defaults to **-1**.
- This command results in history revision.

history words *number-pat event*

Returns the value of the event given by *event*.
- *event* defaults to **-1**.

I

if

Description
Executes scripts conditionally.

Syntax
if *expr1* [**then**] *body1* [**elseif** *expr2* [**then**] *body2*] [**elseif** ...] [**else**] *bodyN*

expr1	Specifies the expression that is evaluated.
body1	Executed by passing it to the PowerTerm PSL engine if *expr1* is true.
expr2	Specifies the expression that is evaluated.
body2	Executed by passing it to the PowerTerm PSL engine if *expr2* is true.

Notes
The **if** command evaluates *expr1* as an expression (in the same way that the **expr** command evaluates its argument). The value of the expression must be a boolean (a numeric value, where 0 is false and anything else 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 PowerTerm PSL engine. 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, 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.

Returns
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 were non-zero and there was no *bodyN*.

Example
Displays "yes" if variable *host* equals *vax*, or "no" if it is not:

```
if {$host == "vax"} {message "yes"} else {message "no"}
```

incr

Description
Increments the value of a variable.

Syntax
incr *varName next*

varName Specifies the name of the variable.
next Specifies the increment for the loop variable.

Notes
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.

Returns
Returns the new *varName* value.

Example
Uses the **incr** command to increase **for** loop counter:

```
for {i = 0} {$i < 10} {incr i} {commands...}
```

indfile

Description
Transfers an IND$FILE file.

Syntax
indfile (*put* | *get* | *set*) *local-file remote-file* [*cms* | *tso* | *cics*] [*ascii yes* | *no*] [*crlf yes* | *no*] [*convert yes* | *no*] [*append* | *overwrite*] [*default* | *fixed* | *variable* | *undefined*] [*tracks* | *cylinders* | *avblocks*] [*lrecl \"size\"*] [*block-size \"size\"*] [*space \"space1\" \"space2\"*] [*options \"options\"*] [*program \"name\"*]

Note Underline indicates the parameter's default value.

File Transfer Action (*put* | *get* | *set*)

put Sends the file from the PC to the host.
get PC receives file from the host.
set Mandatory.
Designates the parameters prior to any file transfer action.

File Names *local-file remote-file*

- Both parameters must be supplied, however they do not have to be identical.
- If the full path is included, then it should appear in quotation marks. For example: "*c:\\license.txt"*
- The name of the file refers to that which will be transferred:

Communication Protocol Type options:

- Default: *cms*.

File Conversion options:

ascii yes | no
- Specifies converting the file to ASCII format.
- Default: no.

crlf yes | no
- For ASCII files (but not for binary files), CR/LF processing is typically appropriate.
- Specifies deleting a carriage return character and a linefeed character from the end of each line of the file you are sending, or adding them to the end of each line of the file you are receiving from the host.
- Default: no.

convert yes | no
- Specifies converting the file according to the format specifications that appear in the ptdef.pts file.
- If the *convert* parameter is set to *yes*, then the following arguments will be referenced from this file:

```
File-Transfer-Host-Data-Type = Host
File-Transfer-PC-Data-Type   = Windows
File-Transfer-Convert-Data   = Off
```

- Default: no

File Creation options:

append
- Default
- Specifies appending the transferred file onto the existing file of same name.

overwrite Specifies that the transferred file overwrite the existing file of same name.

Record Format options:

default
- Default.
- Specifies the **Default** *Record Format* for the file residing on the mainframe.

fixed Specifies the **Fixed** *Record Format* for the file residing on the mainframe.

variable Specifies **Variable** *Record Format* for the file residing on the mainframe.

undefined Specifies **Undefined** *Record Format* for the file residing on the mainframe.

Allocation Units options:

The primary and secondary space allocations are measured in track, cylinder, or average block units. Each type is represented by one of the following fields:

tracks
- Default.
- Specifies **Tracks** as the allocation unit of disk space.

cylinders Specifies **Cylinders** as the allocation of disk space.

avblocks Specifies the size (in blocks) for an average block. This is relevant only where you are using blocks as your allocation unit.

Other options:

[*lrecl "size"*] (Logical Record Length)
- Specifies the record size (in bytes) for the file being created on the host. For ASCII files, set this value to accommodate the longest line in your file.
- Range of values: 0 and 32768
- Range default value: 0
- Lines default: lines of up to 80 characters.

[*block-size "size"*]
- Specifies the block size (in bytes) for the file being created on the host. For files with fixed-length records, this value must be a multiple of the LRECL (Logical Record Length)**.**
- Default value: 0

[*space "space1" "space2"*]
- Specifies the size (in allocation units) of the primary and secondary allocation for the host file being created. In the event that the primary allocation is insufficient, then a secondary allocation is used.
- If the default is not used, then both primary and secondary allocations must be specified.

options Specifies any parameters specific to the IND$FILE program on your host system. The value of the parameter is attached to the end of the transfer command.

program Specifies the name of the host program to be utilized by PowerTerm to initiate a file transfer.

Notes

All of the commands parameters can be modified in the ptdef.pts file under the [3270 IND$FILE] heading.

Returns

If the command succeeds the return value is *True* otherwise it is *False*.

Example

Receives a file named *xyz.a* from the host and renames it *ab.psl* on the PC:

```
indfile get ab.psl xyz.a
```

Receives a file named *curbudgt.a,* from the host and renames it *ltbudgt.a* and converts it to an ASCII format on the PC. *cms* is the communication protocol implemented. File conversion occurs when the file is appended onto the existing file of the same name. The record format is *fixed* and its allocation unit is *tracks*. The *length of the record* is 4000 while its *block size* is 8000. Its *primary allocation size* for the host file being created is 1, while the *secondary allocation size* is 2:

```
indfile put curbudgt.a  ltbudgt.a cms ascii yes crlf yes convert
yes append fixed tracks lrecl "4000" block-size "8000" space "1"
"2"
```

info

Description

Returns information about the state of the PSL interpreter.

Syntax

info *option* [*arg arg ...*]

option Specifies the mode for the command.

Notes

This command provides information about various internals of the PSL interpreter.
The legal *options* (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 PSL command procedure.

info body *procname* — Returns the body of procedure *procname*. *procname* must be the name of a PSL 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 is not specified, returns a list of names of all the PSL commands, including both the built-in commands and the command procedures defined using the **proc** command. If pattern is specified, only those names matching pattern are returned. Matching is determined using the same rules as for string match.

info complete *command*

Returns 1 if command is a complete PSL command in the sense of having no unclosed quotes, braces, brackets or array element names, If the command does not 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 is not 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 PSL command procedure and *arg* must be the name of an argument to that procedure. If *arg* does not 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* is not specified, returns a list of all the names of currently defined global variables. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for string match.

info level [*number*]

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

info library

Returns the name of the library directory in which standard PSL scripts are stored. The default value for the library is "lib", but it may be overridden by setting the PSL_LIBRARY environment variable.

info locals [*pattern*]

If *pattern* is not 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 procs [*pattern*]

If *pattern* is not specified, returns a list of all the names of PSL command procedures. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using

	the same rules as for string match.
info script	If a PSL script file is currently being evaluated, then this command returns the name of the innermost file being processed. Otherwise, the command returns an empty string.
info vars [*pattern*]	If *pattern* is not specified, returns a list of all the names of currently visible variables, including both locals and currently visible globals. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for string match.

Returns
Returns the appropriate info value.

Example
Displays "yes" if variable *varName* exists, otherwise "no":

```
if {[info exists varName] == "vax"} {message "yes"} else {message
"no"}
```

input line

Description
Displays a dialog box enabling user input, which is then returned to the script.

Syntax
input-line *groupboxtext titlebar inputstring*

groupboxtext	Specifies the text that appears above the group box.
titlebar	Specifies the text that appears in the dialog box's title bar.
inputstring	Specifies the string, which will be returned to the script.

Return
The *inputstring* is default and can be overwritten by the user.

Example

```
input-line Your Name Verify  John
```

input password

Description
Displays a dialog box, which requests a string, which is represented by asterisks (***).

Syntax

input-password [*groupboxtext*] [*titlebar*]

groupboxtext	Specifies the text that appears above the group box.
titlebar	Specifies the text that appears in the dialog box's title bar.

Return

N/A

Example

```
input-password "Please type your password" Password
```

iscommand

Description

Verifies if the command is a valid PSL command.

Syntax

iscommand *command*

Return

Returns 1 if the command is a valid PSL command, otherwise returns 0.

Example

```
iscommand cd
```

J

join

Description
Creates a string by joining together list elements.

Syntax
join *list* [*joinString*]

list	Specifies the name of the string will be created
joinString	Specifies character that will be separating each element in the list.

Notes
The list argument must be a valid PSL 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.

Returns
Returns the list items separated by *joinString*.

Example
Displays "1#2#3#4#5#6#7":

```
message [join {Jerry's_list} #]
```

K

key

Description

Maps a key to do a certain action.

Syntax

key [**alt+**][**ctrl+**][**shift+**] *pckey option* [*args...*]

Notes

A PC key or a terminal key may be a combination of control keys plus another key.
Performs one of several key operations, depending on option. The legal options are:

key [alt+][ctrl+][shift+] *pckey* Maps a PC key to run a PSL script.
run scriptName

key [alt+][ctrl+][shift+] *pckey* Maps a PC key to execute PSL commands.
do commands

key [alt+][ctrl+][shift+] *pckey* Maps a PC key to send a VT key.
send **[alt+][ctrl+][shift+]** *vtkey*

The VT key above is a Virtual Terminal key, not DIGITAL's VT emulator key.
The available PC keys are:

- A–Z, 0–9
- Special symbol keys: (` - = \ [ ] ; ' , . /)
- Space, Tab, Capslock, Esc, Backspace, Return
- Right, Left, Up, Down arrows
- Insert, Delete, Home, End, Pgup, Pgdn
- F1–F12
- Print, Scroll, Pause
- Numlock, divide, multiply, subtract, add, decimal, enter
- Numpad0–numpad9

The available VT keys are:

- A–Z, 0–9
- Special symbol keys: (` - = \ [ ] ; ' , . /)

- Space, Tab, Capslock, Esc, Backspace, Return
- Right, Left, Up, Down arrows
- Insert, Delete, Home, End, Pgup, Pgdn
- F1–F12
- Print, Scroll, Pause
- Numlock, divide, multiply, subtract, add, decimal, enter
- Numpad0–numpad9
- All the special VT keys (i.e. Hold, Setup, Print, Dial ...)"

Note

Due to support for several terminals, not all keys support all terminal keyboards, yet it is quite easy to find the name of a requested key in any terminal keyboard.

Returns

Returns an empty string.

Examples

The following line maps ctrl+alt+f5 to activate Notepad.exe:

```
key ctrl+alt+f5 do {exec notepad.exe}
```

The following line maps Shift+Alt+A to run script Menu.psl:

```
key shift+alt+a run menu.psl
```

The following line maps the Scroll Lock key to send “do”:

```
key scroll send do
```

The following line maps Alt+F6 to send Ctrl+g:

```
key alt+f6 send ctrl+g
```

L

lappend

Description
Appends list elements onto a variable.

Syntax
lappend *varName value* [*value value ...*]

varName Specifies the name of the variable, which will have the arguments appended to it.

Notes
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 does not 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 "a = [**concat** $a [**list** $b]]" when $a is long.

Returns
Returns the concatenated list.

Example
Displays "a b {1 2 3} d {1 2 3} 4 5":

```
a = {a b {1 2 3} d}
message [lappend a {1 2 3} 4 5]
```

lindex

Description
Retrieves an element from a list.

Syntax
lindex *list index*

list Specifies the array that will be treated as a PSL list.

index Specifies the number of the list element to be retrieved.

Notes

This command treats *list* as a PSL list and returns the lindex'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 PSL command interpreter; however, variable substitution and command substitution do not occur.

Returns

Returns the specified item. If **lindex** is negative or greater than or equal to the number of elements in value, then an empty string is returned.

Example

Displays the second item from a list in variable List:

```
message [lindex $List 2]
```

linsert

Description

Inserts elements into a list.

Syntax

linsert *list index element* [*element element* ...]

list Specifies the array that will be treated as a PSL list.
index Specifies the number of the list element to be retrieved.
element Specifies the string that will be inserted into the list.

Notes

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

Returns

Returns the new list with old and new items.

Example

Inserts items "vax {hp digital ibm} dg" after item 3 in List:

```
message [linsert $List 3 vax {hp digital ibm} dg]
```

list

Description
Creates a list.

Syntax
list *arg* [*arg arg ...*]

arg Specifies the elements that will comprise the list.

Notes
This command returns a list comprised of all the *args*. Braces and backslashes are 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 *arg* 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.

Returns
Returns the new list.

Examples
Returns "a b {c d e} {f {g h}}":

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

Returns "a b c d e f {g h}":

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

llength

Description
Counts the number of elements in a list.

Syntax
llength *list*

list Specifies the array that will be treated as a PSL list.

Returns
Returns a decimal string giving the number of elements in the list.

Example
Returns 4:

```
llength {1 2 {ab cd fff} {18 19}}
```

lock columns

Description

Ignores any host command that attempts to modify the number of columns in the display area window.

Syntax

lock-columns *number of columns*

number of columns Specifies the number of columns in the display area window, which you desire to lock.

Returns

N/A

Example

Locks 5 columns in the display area window:

```
lock-columns 5
```

lrange

Description

Returns one or more adjacent elements from a list.

Syntax

lrange *list first last*

list Specifies the array that will be treated as a PSL list.
first Specifies the first element in the array.
last Specifies the last element in the array.

Notes

list must be a valid PSL list. This command will return a new list consisting of elements *first* through *last*, inclusive. *last* may be *end* (or any abbreviation of it) to refer to the last element of the list. If *first* is less than zero, it is treated as if it were zero. If *last* is greater than or equal to the number of elements in the *list*, then it is treated as if it were *end*. If *first* is greater than *last*, then an empty string is returned.
lrange *list fiirst first* does not always produce the same result as **lindex** *list first* (although it often does for simple fields that are not enclosed in braces). It does, however, produce exactly the same results as *list* [**lindex** *list first*].

Returns

Returns the items in the range.

Example
Returns "3 {3 4 5} a { b c}":

```
lrange "1 2 3 {3 4 5} a { b c} d" 2 5
```

lreplace

Description
Replaces elements in a list with new elements.

Syntax
lreplace *list first last* [*element element ...*]

list	Specifies the array that will be treated as a PSL list.
first	Specifies the first element in array.
last	Specifies the last element in array.
element	Specifies the replacement of an element in the array.

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

Returns
Returns the new list.

Example
Returns "1 2 one two d":

```
lreplace "1 2 3 {3 4 5} a { b c} d" 2 5 one two
```

lsearch

Description
Checks to see if a list contains a particular element.

Syntax

lsearch [*mode*] *list pattern*

mode	Specifies the method in which the search is carried out.
list	Specifies the array that will be treated as a PSL list.
pattern	Specifies an order of the elements in the array to search for.

Notes

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 *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	pIattern is a glob-style pattern, which is matched against each list element using the same rules as the **string match** command. This is **default**.
-regexp	*pattern* is treated as a regular expression and matched against each list element using the same rules as the **regexp** command.

Returns

Returns the index of the first matched item or -1.

Example

Will find the item "unix" and return 1:

```
lsearch "vax unix ibm" *n*
```

lsort

Description

Sorts the elements of a list.

Syntax

lsort [*switches*] *list*

switches	Specifies the particular sort method used.
list	Specifies the array that will be treated as a PSL list.

Notes

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 switches may be specified before *list* to control the sorting process (unique abbreviations are accepted):

-ascii	Uses string comparison with ASCII collation order. This is the **default**.
-integer	Converts list elements to integers and uses integer comparison.
-real	Converts list elements to floating-point values and uses floating comparison.
-command *command*	Uses **command** as a comparison command. To compare two elements, evaluate a PSL 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	Sorts the list in increasing order ("smallest" items first). This is the **default**.
-decreasing	Sorts the list in decreasing order ("largest" items first).

Returns

Returns the sorted list.

Example

Returns "ibm unix vax":

```
lsort "vax unix ibm"
```

M

md

Description
Creates a working folder.

Syntax
md *dirName*

dirName Specifies the name of the folder being created.

Notes
In the event that a path is not specified then the folder will be created under the PowerTerm directory.

Returns
Returns an empty string.

menu

Description
Changes menu type.

Syntax
menu *option*

option Specifies the action to be done to the menu.

Notes
Changes PowerTerm main menu type:

menu hide Removes the main menu.

menu minimize Minimizes the main menu (with one click on the menu, you can

restore it).

menu restore Restores the menu to its normal state.

Example

Hides the menu:

```
menu hide
```

message

Description

Displays a message.

Syntax

message *text* [*option*]

text Specifies the text that appears in the message box's title bar.
option Specifies what kind of message box will be displayed.

Notes

Displays the message string on the screen. The options are:

ask A *Question mark* message box with *Yes/No* buttons is displayed.

- **1** indicates that the *Yes* button was pressed.
  0 indicates that the *No* button was pressed.
- a="Do you want to close your session?"
  message $a ask

error An *Error* message box is displayed.

- If the command succeeds, the return value is *True* otherwise it is *False*.
- a="The file opened cannot be opened"
  message $a error

info An *Information* message box is displayed.

- If the command succeeds, the return value is *True* otherwise it is *False*.
- a="The host is being shutdown for maintenance"
  message $a info

information An *Information* message box is displayed.

- If the command succeeds, the return value is *True* otherwise it is *False*.
- a="The host is being shutdown for maintenance!"
  message $a information

okcancel

A *Question* mark message box with *OK/Cancel* buttons is displayed.

- **1** indicates that the *OK* button was pressed.
  0 indicates that the *Cancel* button was pressed.
- a=”Do you want to close your session?”
  message $a okcancel

question

A *Question mark* message box with OK/Cancel is displayed.

- **1** indicates that he *OK* button was pressed.
  0 indicated that the *Cancel* button was pressed.
- a=”Do you want to close your session?”
  message $a question

warn

A *Message* box with warning sign is displayed.

- If the command succeeds, the return value is *True* otherwise it is *False*.
- a=”Please logoff now!”
  message $a warn

warning

A *Message* box with warning sign is displayed.

- If the command succeeds, the return value is *True* otherwise it is *False*.
- a=”Please logoff now!”
  message $a warning

yesno

A *Question mark* message box with Yes/No is displayed.

- **1** indicates that the *Yes* button was pressed
  0 indicates that the *No* button was pressed.
- a=”Do you want to close your session?”
  message $a yesno

Example

Displays the message "Hello" with title "Welcome" and message type info:

```
message Hello title Welcome info
```

move file

Description

Moves a file from its present location to a designated location.

Syntax

move-file *old-location new-location*

old-location Specifies the former location of a designated file.
new-location Specifies the new location of a designated file.

Note

If the same name of the file being moved exists in the *new-location*, it will be overwritten.

Return

N/A

Example

Moves the file *current_data.txt* from its present location on the C drive to the same folder on the D drive:

```
move-file //C: Program Files/Ericom/WebConnect/ current_data.txt
//D: Program Files/Ericom/WebConnect/ current_data.txt
```

O

open

Description
Opens a file.

Syntax
open *fileName* [*access*] [*permissions*]

fileName	Specifies the name of the file to be opened.
access	Specifies what can be done to the designated file.
permissions	Specifies with which permissions the file will be opened.

Notes
This command opens a file and returns an identifier that may be used in future invocations of commands like **read**, **gets**, **puts**, and **close.**
The *access* argument indicates the way in which the file is to be accessed. It may take two forms:

- string, or
- a list of POSIX access flags.

In the string form, *access* may have any of the following values:

r	Opens the file for reading only. The file must already exist. This is default value.
r+	Opens the file for both reading and writing. The file must already exist.
w	Opens the file for writing only. Truncates the file if it exists. If it does not exist, it creates a new file.
w+	Opens the file for reading and writing. Truncates the file if it exists. If it does not exist, it creates a new file.
a	Opens the file for writing only. The file must already exist. The file is positioned so that new data is appended to the file.
a+	Opens the file for reading and writing. If the file does not exist,

it creates a new empty file. It sets 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. This parameter is only relevant if you select *w*.

RDONLY Opens the file for reading only.

RDWR Opens the file for both reading and writing.

Returns
Returns the file ID.

Example
Opens file "output.dat" for writing, writes data and closes:

```
File = [open output.dat w]
pets $File "message test"
close $File
```

open keyboard file

Description
Opens keyboard file.

Syntax
open-keyboard-file *file-name*

fileName Specifies the name of the keyboard file to be opened.

Notes
Supports logical directories. If the filename has no path, is supposed to refer to **<connections>\filename**.

Returns
Returns an empty string.

Example
Opens keyboard file *keyboard.kbd:*

```
open-keyboard-file keyboard.kbd.
```

open power pad file

Description
Opens Power Pad file.

Syntax
open-power-pad-file *file-name*

fileName Specifies the name of the Power Pad file to be opened.

Notes
Supports logical directoriesLogical_directories. If the filename has no path, is supposed to refer to **<connections>\filename**.

Returns
Returns an empty string.

Example
Opens keyboard file *mypad.pad:*

```
open-power-pad-file mypad.pad.
```

open setup file

Description
Opens setup file.

Syntax
open-setup-file *file-name*

file-name Specifies the name of the setup file to be opened.

Notes
Supports logical directoriesLogical_directories. If the filename has no path, is supposed to refer to **<connections>\filename**.

Returns
Returns an empty string.

Example
Opens setup file "prog.pts":

```
open-setup-file prog.pts
```

P

pad

Description
Defines the keys on the PowerPad to run any PSL command.

Syntax
pad *row col description {}*

row	Specifies in which row of the Power Pad to assign the designated description.
col	Specifies in which column of the Power Pad to assign the designated description.
description	Specifies the text that appears on the designated Power Pad button.

Return
N/A

Example
```
pad 1 2 F2 {message "Hello"}
```

paste

Description
Pastes the specified text to the designated screen location.

Syntax
paste *col row text*

col	Specifies in which column on the screen to paste the text.
row	Specifies in which row on the screen to paste the text.
text	Specifies the string, which will be pasted to the designated screen location.

Notes
The screen location (column and row), which you designate for the text to appear, must be an unprotected field space. If the specified text is more than one word long then it must be enclosed in quotation marks.

NOTE *Only applicable in an IBM–block mode application.*

Returns
N/A

Example
```
paste 10 55 "Good Morning"
```

paste from clipboard

Description
Deposits current clipboard contents into emulation window where the cursor is positioned.

Syntax
paste-from-clipboard

Returns
N/A

print file

Description
Prints the specified files to the operating system's default printer.

Syntax
print-file *filename A filename B…*

filename Specifies the name of the file to be printed.

Notes
Multiple files can be designated for the printer by listing them and leaving a space in between each file name.

Returns
N/A

Example
```
print-file lastyears_budget.txt currentyears_budget.txt
```

print screen

Description
Prints the contents of the work area or the selected text.

Syntax
print-screen

Notes
Simulates the functionality of the Print Screen toolbar button.

Returns
1 if successful and 0 if not.

proc

Description
Creates a PSL procedure.

Syntax
proc *name args body*

name Specifies the name of a new PSL procedure.
args Specifies the formal parameters to the procedure.
body Specifies the procedures main section of commands.

Notes
The proc command creates a new PSL 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 PSL interpreter. *args* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of those 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 do not 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.
When a procedure is invoked, the procedure's return value is the value specified in a return command. If the procedure does not 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.

Returns
Returns an empty string.

Example
Defines a recursive factorial procedure:

```
proc factorial x {
      if {$x == 1} {return 1}
      return [expr {$x * [factorial [expr $x - 1]]}]
}
```

Will display 24:

```
message [facorial 4]
```

puts

Description
Writes to a file.

Syntax
puts [**-***nonewline*] *fileId string*

nonewline	Specifies whether to suppress a new line character.
fileId	Specifies a file that was opened for writing.
string	Specifies the procedures main section of commands.

Notes
Writes the characters given by *string* to the file given by *fileId*. *fileid* must have been the return value from a previous call to **open**.
puts normally outputs a newline character after *string*, but this feature may be suppressed by specifying the **-nonewline** switch. Output to files is buffered internally by PSL. The **flush** command may be used to force buffered characters to be output.

Returns
Returns an empty string.

Example
Opens file *data* for writing, writes data, and closes:

```
outFile = [open data w]
```

```
puts $outFile Information
close $outFile
```

pwd

Description
Returns the current working directory.

Syntax
pwd

Returns
Returns the path name of the current working directory.

Example
Displays the current working directory:

```
message [pwd]
```

R

read

Description
Reads from a file.

Syntax
read [-*nonewline*] *fileId*
or
read *fileId numBytes*

nonewline	Specifies whether to suppress a new line character.
fileId	Specifies a file that was opened for writing.
numBytes	Specifies how many bytes to read.

Notes
- In the first form, all of the remaining bytes are read from the file given by *fileId*. They are returned as the result of the command. If the **-nonewline** switch is specified, then the last character of the file is discarded if it is a newline.
- In the second form, the extra argument specifies how many bytes to read. Exactly this many bytes will be read and returned, unless there are fewer than *numBytes* bytes left in the file. In this case, all the remaining bytes are returned. *fileid* must be the return value from a previous call to **open** and it must refer to a file that was opened for reading.

Returns
Returns the data read from the file.

Example
Opens file *data* for reading, reads 10 bytes, and closes:

```
inFile = [open data]
data = [read $inFile 20]
close $inFile
```

recv ascii file

Description
Receives an ASCII file from the host.

Syntax
recv-ascii-file *filename method displaycharacteristic*

filename	Specifies the file being transferred.
method	Specifies what to do with the received file.
displaycharacteristic	Specifies whether to display the transferred data.

Notes
The *method* argument can receive the value *append,* while *displaycharacteristic display-data.*
Available communication protocol types:
- kermit
- xmodem
- ymodem
- zmodem

Returns
An empty string.

Example
```
recv-ascii-file login.psl append display-data
```

recv binary file

Description
Receives a binary file to the host.

Syntax
recv-binary-file *commprotocol filename*

commprotocol	Specifies the communication protocol being implemented in this file transfer.
filename	Specifies the file being transferred.

Notes
Available communication protocol types:
- kermit
- xmodem
- ymodem
- zmodem

Returns
Returns an empty string.

Example
```
recv-binary-file kermit login.psl
```

recv binary stop

Description
Stops PowerTerm from receiving the ASCII file.

Syntax
recv-binary-stop *commprotocol filename*

commprotocol	Specifies the communication protocol being implemented in this file transfer.
filename	Specifies the file being transferred.

Notes
Available communication protocol types:
- kermit
- xmodem
- ymodem
- zmodem

Returns
Returns an empty string.

Example
```
recv-binary-stop kermit logoff.psl
```

recv file

Description
Receives a file from the host.

Syntax
recv-file *commprotocol filename*

commprotocol	Specifies the communication protocol being implemented in this file transfer.
filename	Specifies the file being transferred.

Notes

When you execute this command, PowerTerm opens a submenu that lists the transmission options. Each option sends files using the protocol that the option represents. All file transmission options lead to the same dialog box, in which you select the file to be sent.

Permissible communication protocol types:

- kermit
- xmodem
- ymodem
- zmodem

Returns

Returns an empty string.

Example

```
recv-file kermit runbudget.psl
```

recv indfile

Description

Receives an *ind$file* file from the host to the PC.

Syntax

recv-indfile

Notes

Only for 3270 emulations.

This command can be executed successfully only if the relative parameters are previously designated in either:

- The *ptdef.pts* file, under the **[3270 IND$FILE]** heading.
- The **Ind$File: Receive File** dialog box (accessed by **Communication | Receive File**).
- The **indfile** command.

When you select this command, PowerTerm opens a dialog box with tabs that correspond to the transmission options. Each option receives files using the protocol that the option represents.

Returns

N/A

regexp

Description

Matches a regular expression against a string.

Syntax

regexp [*switches*] *exp string* [*matchVar*] [*subMatchVar subMatchVar ...*]

switches	Specifies the command mode.
exp	Specifies the expression that is compared to a string.
string	Specifies the string to be compared to a designated expression.
matchVar	Set to the range of *string* that matches all of *exp.*
subMatchVar	Contains the characters in *string* that matches the leftmost parenthesized subexpression within *exp.*

Notes

Determines whether the regular expression *exp* matches part or all of *string.*
If additional arguments are specified after *string*, then they are treated as the names of variables in which to return information about which part(s) of *string* matched *exp*.
matchVar will be set to the range of *string* that matched all of *exp*. The first sub *matchVar* will contain the characters in *string* that matched the leftmost parenthesized subexpression within *exp*. The next *submatchVar* will contain the characters that matched the next parenthesized subexpression to the right in *exp*, and so on.
If the initial arguments to **regexp** start with "-", then they are treated as switches. The following switches are supported:

-nocase	Causes uppercase characters in *string* to be treated as lower case during the matching process.
-indices	Changes what is stored in the *submatchVar*'s. Instead of storing the matching characters from *string*, each variable will contain a list of two decimal strings giving the indices in *string* of the first and last characters in the matching range of characters.
--	Marks the end of switches. The argument following this one will be treated as *exp* even if it starts with a "-".

If there are more *subMatchVar*'s than parenthesized subexpressions within *exp*, or if a particular subexpression in *exp* does not match the string (for example, because it was in a portion of the expression that was not matched), then the corresponding *subMatchVar* will be set to "-1 -1" if **-indices** has been specified or to an empty string otherwise.

Regular Expressions

- A *regular expression* is zero or more branches, separated by "|". It matches anything that matches one of the branches.
- A *branch* is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, and so on.
- A *piece* is an atom possibly followed by "*", "+", or "?".
  If it is followed by "*" then it matches a sequence of zero or more matches of the atom.
  If it is followed by "+" then it matches a sequence of 1 or more matches of the atom.
  If it is followed by "?" then it matches a match of the atom, or the null string.
- An *atom* can be:
  A regular expression in parentheses, matching a match for the regular expression.
  A range (see below).

A dot (.), matching any single character.
A caret (^), matching the null string at the beginning of the input string.
A dollar sign ($), matching the null string at the end of the input string.
A backslash (\) followed by a single character, matching that character, or a single character with no other significance, matching that character.

- A *range* is a sequence of characters enclosed in brackets [ ]. It normally matches any single character from the sequence.
  If the sequence begins with "^", it matches any single character not from the rest of the sequence.
  If two characters in the sequence are separated by "-", this is shorthand for the full list of ASCII characters between them. For example, "[0–9]" matches any decimal digit.
  To include a literal "]" in the sequence, make it the first character, following a possible "^".
  To include a literal "-", make it the first or last character.

Choosing Among Alternative Matches

In general there may be more than one way to match a regular expression to an input string. For example, consider the following command:

```
regexp (a*)b* aabaaabb x y
```

Considering only the rules given so far, *x* and *y* could end up with the values *aabb* and *aa*, *aaab* and *aaa*, *ab* and *a*, or any of several other combinations. To resolve this potential ambiguity, **regexp** chooses among alternatives using the rule *"first then longest"*. In other words, it considers the possible matches in order, working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

Rule 1	If a regular expression could match two different parts of an input string, then it will match the one that begins *earliest*.
Rule 2	If a regular expression contains "|" operators, then the *leftmost* matching subexpression is chosen.
Rule 3	In "*", "+", and "?" constructs, *longer* matches are chosen in preference to shorter ones.
Rule 4	In sequences of expression components, the components are considered *from left to right*.

In the example from above, *(a*)b** matches *aab*. The *(a*)* portion of the pattern is matched first and it consumes the leading *aa*, then the *b** portion of the pattern consumes the next *b*.

Returns

Returns 1 if it matches, 0 if it does not.

Example

After this command, *x* will be *abc*, *y* will be *ab*, and *z* will be an empty string.
Rule 4 specifies that *(ab|a)* gets first shot at the input string, and Rule 2 specifies that the *ab* subexpression is checked before the *a* subexpression. Thus the *b* has already been claimed before the *(b*)* component is checked, and *(b*)* must match an empty string.

```
regexp (ab|a)(b*)c abc x y z
```

regsub

Description

Performs substitutions based on regular expression pattern matching.

Syntax

regsub [*switches*] *exp string subSpec varName*

switches	Specifies the command mode.
exp	Specifies the expression that is compared to a string.
string	Specifies the string to be compared to a designated expression.
subSpec	Specifies special execution of the command.
varName	Specifies the name of the variable, which has string copied to it.

Notes

This command matches the regular expression *exp* against *string*, and it copies *string* to the variable whose name is given by *varName*. If there is a match, then while copying *string* to *varName*, the portion of *string* that matched *exp* is replaced with *subSpec*. If *subSpec* contains a "*&*" or "*0*", then it is replaced in the substitution with the portion of *string* that matched *exp*. If *subSpec* contains an "*n*", where *n* is a digit between 1 and 9, then it is replaced in the substitution with the portion of *string* that matched the *n*-th parenthesized subexpression of *exp*. Additional backslashes may be used in *subSpec* to prevent special execution of "*&*" or "*0*" or "*n*" or *backslash*.

NOTE The use of backslashes in subSpec *tends to interact badly with the PSL parser's use of backslashes, so it is generally safest to enclose* subSpec *in braces if it includes backslashes.*

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

-all	All ranges in *string* that match *exp* are found and substitution is performed for each of these ranges. Without this switch, only the first matching range is found and substituted. If **-all** is specified, then "*&*" and "*n*" sequences are handled for each substitution, using the information from the corresponding match.
-nocase	Uppercase characters in *string* will be converted to lowercase before matching against *exp*. However, substitutions specified by *subSpec* use the original unconverted form of *string*.
--	Marks the end of switches. The argument following this one will be treated as *exp* even if it starts with a "-".

Returns

Returns 1 if it matches, 0 if it does not.

Example

Returns 1 and variable *new* will contain *"*abc*abc*def"*. The *abc* portion of *abcdef* was substituted by **abc*abc** and the *def* portion was left at the end:

```
regsub (ab|a)(b*)c abcdef *&*&* new
```

remove menu item

Description
Removes the specified menu item.

Syntax
remove-menu-item *menu-item*

Notes
Once a menu item is removed, it cannot be restored.
For a complete list of menu items, see Menu Items.

Returns
N/A

rename

Description
Renames or deletes a command.

Syntax
rename *oldName newName*

oldName Specifies the old name of a file.
newName Specifies the new name of a file.

Notes
Renames the command that used to be called *oldName* so that it is now called *newName*. If *newName* is an empty string, then *oldName* is deleted.

Returns
Returns an empty string as result.

Example
Renames the for command to loop and then uses it:

```
rename for loop
      loop {i = 0} {$i < 10} {incr i} {commands...}
```

rename-file

Description
Renames a file.

Syntax
rename-file *old-filename new-filename*

old-filename	Specifies the old name of a file.
newName	Specifies the new name of a file.

Note
If the *new-filename* exists, it will be overwritten.

Return
N/A

Example
Renames the file named "2000_budget.txt" to "Current_budget.txt":

```
rename-file 2000_budget.txt Current_budget.txt
```

return

Description
Returns from a procedure.

Syntax
return [*-code code*] [*-errorcode code*]

-code	Specifies that procedures that implement new control structures can reflect exceptional conditions back to their callers.
errorcode	Specifies the new name of a file.
code	Contains a value for the *errorcode* variable.

Notes
Returns immediately from the current procedure, top-level command, or **run** 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 is not specified, the procedure will return normally. 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. The same as if the option is omitted.
error	*Error* return. The same as if the **error** command

	were used to terminate the procedure, except for handling of *errorInfo* and *errorCode* variables.
return	The current procedure will return with a completion code of **PSL_RETURN**, so that the procedure that invoked it will also return.
break	The current procedure will return with a completion code of **PSL_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 **PSL_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.

An additional option, **-***errorcode*, may be used to provide additional information during error returns. This option is ignored unless *code* is *error*.
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*.

Returns

Returns the string specified or an empty string.

Example

Defines a division procedure with two parameters. If the second parameter is *zero*, it returns with **continue** error code. Otherwise, it returns with division of the two.
In the rest of the code, in a loop: inputs two numbers and activates the divide procedure. If the second parameter is zero, *divide* will act as a **continue** command and return to start of the **while** loop. Otherwise, the division result will be displayed and the loop will be terminated:

```
proc divide {x y} {
      if {$y == 0} {return -code continue 0} else {
      return [expr $x / $y]}
}
while {1} {
      #     commands to input data for variables x & y ...
            result = [divide $x $y]
            message "$x / $y = $result"
            break
}
```

ring bell

Description
Rings a bell.

Syntax
ring-bell

Return
N/A

run

Description
Evaluates a file as a PSL script.

Syntax
run *fileName* [*parameters...*]

fileName	Specifies the name of the file to be evaluated as a PSL script.
parameters	Specifies the various parameters that will be used in this script.

Notes
Reads file *fileName* and pass the contents to the PSL interpreter as a script to evaluate in the normal fashion. The return value from **run** is the return value of the last command executed from the file. If an error occurs in evaluating the contents of the file, then the **run** command will return that error. If a **return** command is invoked from within the file, then the remainder of the file will be skipped and the **run** command will return normally with the result from the **return** command. If parameters are included, variables named *$p1* to *$pN* will exist, according to number of parameters.

Supports logical directories . If the filename has no path, it is supposed to refer to <*connections*>*filename*.

Returns
Returns the value from the last command in the script.

Example
```
run test.psl
```

S

save keyboard file

Description
Saves the keyboard mapping file (.**kbd**).

Syntax
save-keyboard-file

Returns
N/A

save power pad file

Description
Saves the power-pad file (**.pad**).

Syntax
save-power-pad-file

Returns
N/A

save setup file

Description
Saves the terminal settings file (**.pts**).

Syntax
save-setup-file

Returns

N/A

scan

Description

Parses a string using conversion specifiers in the style of *sscanf*.

Syntax

scan *string format varName* [*varName ...*]

string	Specifies the name of the input string to be parsed.
format	Specifies how to parse the string.
varName	Specifies the name of a variable to which a string is assigned.

Notes

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 fields successfully parsed.
string gives the input to be parsed and *format* indicates how to parse it, using % conversion specifiers as in *sscanf*. Each *varName* gives the name of a variable. When a field is scanned from *string*, the result is converted back into a string and assigned to the corresponding variable.

Details on Scanning

scan operates by scanning str*i*ng and *formatString* together. If the next character in *formatString* is a blank or tab, then it is ignored. Otherwise, if it is not a % character, then it must match the next non-white-space character of string. When a % is encountered in *formatString*, it indicates the start of a conversion specifier.

A conversion specifier contains three fields after the %, which all are optional except for the conversion character:

- Indicates that the converted value is to be discarded instead of assigned to a variable.
- A number, indicating a maximum field width.
- A conversion character.

When **scan** finds a conversion specifier in *formatString*, it first skips any white-space characters in *string*. Then it converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to scan. The following conversion characters are supported:

d	The input field must be a *decimal integer*. It is read in and the value is stored in the variable as a decimal string.
o	The input field must be an *octal integer*. It is read in and the value is stored in the variable as a decimal string.

x	The input field must be a *hexadecimal integer*. It is read in and the value is stored in the variable as a decimal string.
c	A *single character* is read in and its binary value is stored in the variable as a decimal string. Initial white space is not skipped in this case, so the input field may be a white-space character. This conversion is different from the ANSI standard in that the input field always consists of a single character and no field width may be specified.
s	The input field consists of all the characters up to the next white-space character. The characters are copied to the variable.
e or **f** or **g**	The input field must be a floating-point number consisting of an optional sign, a string of decimal digits possibly 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.
[^chars]	The input field consists of any number of characters not in *chars*. The matching string is stored in the variable. If the character immediately following the "^" is a "]", then it is treated as part of the set rather than the closing bracket for the set. The number of characters read from the input for a conversion is the largest number that makes sense for that particular conversion. For example, 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.

Returns

Returns the number of scanned parameters and fills the variables with their values.

Example

Scans the following string and sets the variables:

```
var1 = 123  var2 = 65 ("A" ASCII)  var3 = unix.
scan "123 A unix" "%d %c %s" var1 var2 var3
```

screen

Description

Copies data from the screen.

Syntax

screen *startRow startCol* [*endRow*] *endCol*

startRow	Specifies the *uppermost* row of the required text area of the emulation screen.
startCol	Specifies the *leftmost* column of the required text area of the emulation screen.
endRow	Specifies the *lowermost* row of the required text area of the emulation screen.
endCol	Specifies the *rightmost* column of the required text area of the emulation screen.

Notes

screen copies complete lines from the starting position (*startRow*, *startCol*) to and including the end position (*endRow*, *endCol*).
If *endRow* is not specified, *endRow* equals *startRow*.

Returns

Returns the data copied from the screen.

Example

Consider the following screen:

Line 1:	"line 1: 1234567890"
Line 2:	"line 2: 1234567890"
Line 3:	"line 3: 1234567890"
Line 4:	"line 4: 1234567890"

```
data = [screen 2 15 4 10]
```

Sets variable data to screen data from (2, 15) to (4, 10):

```
"7890\nline 2: 1234567890\nline 3: 12"
```

where "\n" is a line separator.

screen-rect

Description

Copies data from the screen.

Syntax

screen-rect *startRow startCol* [*endRow*] *endCol*

startRow	Specifies the *uppermost* row of the required text area of the emulation screen.
startCol	Specifies the *leftmost* column of the required text area of the emulation screen.
endRow	Specifies the *lowermost* row of the required text area of the emulation screen.
endCol	Specifies the *rightmost* column of the required text area of the emulation screen.

Notes

screen-rect copies rectangular data from the starting position (*startRow*, *startCol*) to and including the end position (*endRow*, *endCol*).
If *endRow* is not specified, *endRow* equals *startRow*.

Returns

Returns the data copied from the screen.

Example

Consider the following screen:
line 1: 1234567890
line 2: 1234567890
line 3: 1234567890
line 4: 1234567890

```
data = [screen-rect 1  2 3  15]
```

Sets variable data to screen data from (1, 2) to (3, 15):

```
"ine 1: 1234567
 \n
 ine 2: 1234567
 \n
 ine 3: 1234567
 \n"
```

where "\n" is a line separator.

seek

Description

Changes the access position for an open file.

Syntax

seek *fileId offset* [*origin*]

fileId	Specifies the file upon which the current access position is changed.
offset	Specifies the initial access position of the seek command.
origin	Specifies the point of reference in the file from where the *offset* is calculated.

Notes

fileId must be the return value from a previous call to **open**.The *offset* and *origin* arguments specify the position at which the next read or write will occur for *fileId*. *offset* must be an integer (can be negative) and *origin* must be one of the following:

start	The new access position will be *offset* bytes from the start of the file. This is the default.
current	The new access position will be offset bytes from the current access position. A negative *offset* moves the access position backwards in the file.
end	The new access position will be *offset* bytes from the end of the file. A negative *offset* places the access position before the end-of-file, and a positive *offset* places the access position after the end-of-file.

Returns

Returns an empty string.

Example

Opens file *file1* for reading. Moves to position 55 in the file. Reads 10 bytes to variable data and closes file:

```
fileId = [open file1]
seek $fileId 55 start
data = [read $fileId 10]
close $fileId
```

send

Description

Sends data to the host.

Syntax

send *data*

data Specifies the information that is sent to the host.

Notes

Sends *data* as if it is typed on the keyboard. The "^" sign followed by a letter represents the control code of that letter. For example, *^M* sends *ctrl-m*.

Returns

Returns an empty string.

Example

The host receives the following data and shows the current directory:

```
send "dir^M"
```

send abort output

Description

Sends escape sequences to the host to stop sending output.

Syntax

send-abort-output

Notes

Only in Telnet emulations.

Returns

Returns an empty string.

send ascii file

Description

Sends a file as ASCII code to the host.

Syntax

send-ascii-file *filename*

filename Specifies the file that is sent as ASCII code.

Notes

Available communication protocol types:

- kermit
- xmodem
- ymodem
- zmodem

Returns

An empty string.

Example

```
send-ascii-file exit.psl
```

send binary file

Description
Sends a binary file to the host.

Syntax
send-binary-file *commprotocol filename*

commprotocol	Specifies the communication protocol being implemented in this file transfer.
filename	Specifies the file being transferred.

Notes
Available communication protocol types:

- kermit
- xmodem
- ymodem
- zmodem

Returns
An empty string.

Example
```
send-binary-file ymodem register.psl
```

send break

Description
Sends a break command to the host.

Syntax
send-break

Notes
Simulates a user pressing the break key.

Returns
Returns an empty string.

send file

Description

Sends a file from the PC to the host.

Syntax

send-file *commprotocol filename*

commprotocol	Specifies the communication protocol being implemented in this file transfer.
filename	Specifies the file being transferred.

Notes

Permissible communication protocol types:

- kermit
- xmodem
- ymodem
- zmodem

When you execute this command, PowerTerm opens a submenu that lists the transmission options. Each option sends files using the protocol the option represents. All file transmission options lead to the same dialog box, in which you select the file to be sent.

Returns

Returns an empty string.

Example

```
send-file xmodem 1.psl
```

send indfile

Description

Sends a file from the PC to the host.

Syntax

send-indfile

Notes

Only for 3270 emulations.

This command is executed successfully only if the relative parameters are previously designated in either:

- The *ptdef.pts* file under the **[3270 IND$FILE]** heading.
- The *Ind$File: Receive File* dialog box (accessed by **Communication | Receive File**).
- The **indfile** command.

When you execute this command, PowerTerm opens a submenu that lists the transmission options. Each option sends files using the protocol the option represents. All file transmission options lead to the same dialog box, in which you select the file to be sent.

Returns
Returns an empty string.

send raw data

Description
Sends raw data to the host.

Syntax
send-raw-data *data*

data Specifies the data to be sent to the host.

Notes
Sends data exactly as is. No conversion is done.
The "^" sign followed by a letter represents the control code of that letter. *^M* sends *ctrl-m*.

Returns
Returns an empty string.

Example
The host receives the following data and shows the current directory:

```
send-raw-data "dir^M"
```

session

Description
Modifies the communication session status.

Syntax
session *option*

option Specifies the method that the session operation will employ.

Notes
Performs one of several session operations, depending on option. The legal options are:

session open	Opens a session according to communication parameters previously defined with the set command.
session modify	Modifies the current session according to communication parameters previously defined with the set command.

session close Closes the current session.

In order to automatically connect to a host, you have to make the appropriate script and add the name of the script, with parameters, to the Property's Command line. PowerTerm will run the script and open the session every time the icon is clicked.

Returns

Returns an empty string.

Examples

Opens a COM session with the following parameters:

```
set comm-type com
set port-number 2
set baud-rate 19200
set protocol-type xonxoff
session open
```

Modifies the COM session to 9600 baud-rate:

```
set baud-rate 9600
session modify
```

Opens the setup file "abc.pts" for working with specific PowerTerm parameters for the "abc" host (similar to the Open command on the File menu). Then opens a Telnet session to host "abc" (similar to the Connect command on the Communication menu).

```
open-setup-file abc.pts
set comm-type telnet
set host-name abc
session open
```

Opens a lat session to host "abc" through DIGITAL PATHWORKS 32:

```
set comm-type lat
set service-name abc
session open
```

Opens a lat session to host "abc" through Novell's NetWare for LAT:

```
set comm-type lat
set server-name NovellServerName
set service-name abc
session open
```

Closes the current session:

```
session close
```

A script, named *telnet.psl*, is connected to a host with telnet protocol:

1. Click Properties on the File menu, and enter:
   `C:\PTW\PTW.EXE telnet.psl HostName`
2. Create the following script:

```
set comm-type telnet
set host-name $p1
session open
```

Every time you click the icon, *telnet.psl* will execute and connect to *HostName*. In this manner you can create several icons for automatic connection to all your organization's computers.

set

Description

Sets a new value to a PowerTerm parameter.

Syntax

set *parameter value [value2]*

Notes

Sets a new value to one of the PowerTerm parameters.

set buttons [on|off]	Shows/hides the Soft buttons.
set block-paste on | **off**	Enables the block-paste mode algorithm.
set end-of-medium cr | **crlf** | **ff**	Only for IBM emulations. Sets the end-of-medium printing mode.
set func-cols *num-of-columns*	Sets the number of function buttons columns.
set func-hotspot-rows *start end*	Only for IBM emulations. • Sets the range of rows in which function hotspots are enabled. • If the value for *start* or *end* is zero, ignore the parameter. • Negative values can be used in order to back track from the last line of the screen.
set keyboard [**default** | **lk450**]	Sets the keyboard type.
set mark-func-only on | **off**	Enables function hotspots to mark only the function key's name.
set menu-hotspots on | **off**	Only for IBM emulations. Activates the menu hotspots.

set menu-hotspot-rows *start end*	Only for IBM emulations. • Sets the range of rows in which menu hotspots are enabled. • If the value for *start* or *end* is zero, ignore the parameter. • Valid values start from row 1.
set mouse-report on | **off**	Only for VT emulations. Enables mouse report.
set option-hotspots on | **off**	Only for IBM emulations. Activates the option hotspots.
set pad-cols *num-of-columns*	Sets the number of power-pad columns.
set pad-rows *num-of-rows*	Sets the number of power-pad rows.
set pad-pos *left top*	Sets the power-pad position.
set pad-size *width height*	Sets the power-pad size.
set print-device-name *device-name*	Sets the print device name.
set print-direction r2l | **l2r**	Sets the print direction to right-to-left (r2l) or left-to-right (l2r).
set print-file-name *filename*	Sets the print filename.
set print-prefix *text*	Sets the print prefix.
set print-screen-convert no_convert | ibm | digital | graphics	Sets the print screen conversion mode.
set print-suffix *text*	Sets the print suffix.
set printer-transparent-footer [*footer*]	Sets the printer transparent footer.
set printer-transparent-header [*header*]	Sets the printer transparent header.
set repeat-left-alt-key on | **off**	Allows repetition of left-alt key.
set repeat-right-alt-key on | **off**	Allows repetition of right-alt key.
set repeat-left-ctrl-key on | **off**	Allows repetition of left-ctrl key
set repeat-right-ctrl-key on | **off**	Allows repetition of right-ctrl key.

set show-buttons on | off	Shows/hides the function buttons.
set show-status-bar on | off	Shows/hides the status bar.
set show-toolbar on | off	Shows/hides the toolbar.
set slave-printer-convert no_convert | ibm | digital | graphics	Sets the slave printer conversion mode.
set slave-printer-delimiter *decimal-ascii-code*	Determines the job delimiter character that will divide the data into print jobs when printing in slave mode, instead of escape sequences arriving from the host application.
set status-bar on | off	Activates the status bar.
set toolbar on | off	Activates the toolbar.
set vt-hotspots on | **off**	Only for VT emulations. Activates the VT hotspots.
set wyse-esc-b on | **off**	Only for Wyse emulations. Ignores the escape sequence *esc-b*.

Returns
N/A

set auto signon

Description
Tells the transport protocol to automatically log you on to the host once a connection is established.

Syntax
set auto-signon *yes* | *no*

yes	Instructs the transport protocol to automatically log you on to the host once a connection is established.
no	Does not instruct the transport protocol to automatically log you on to the host once a connection is established.

Returns
N/A

set baud rate

Description

Tells the transport protocol to automatically log you on to the host once a connection is established.

Syntax

set baud-rate *number*

number Specifies the value that represents the baud rate to use.

Legitimate values for *number* are:

- 50
- 75
- 110
- 134
- 150
- 300
- 600
- 1200
- 1800
- 2000
- 2400
- 3600
- 4800
- 7200
- 9600
- 14400
- 19200
- 38400
- 57600
- 115200

Returns

N/A

set comm-type

Description

Designates which communication protocol to use.

Syntax

set comm-type *commtypeName*

commtypeName Specifies the value that represents the communication protocol to use.

Legitimate values for *commtypeName* are:

- com
- telnet
- bapi
- cterm
- lat
- tapi
- superlat
- nwlat
- tn3270
- mssna
- nwsaa
- nwsaa-ipx
- nwsaa-tcpip
- tn5250
- appc
- ns-router
- winappc

Returns
N/A

Example
```
set comm-type com
```

set device-name

Description
Specifies the device to which to connect.

Syntax
set device-name *name*

name Specifies the alphanumeric string containing the desired device name.

Notes
Only for AS/400 printing sessions.

Returns
N/A

set disable-exit-active-session

Description
Determine whether or not the terminal will exit after the timeout (which was specified in the terminal setup preferences) has expired.

Syntax
set disable-exit-active-session *on* | *off*

on Specifies that the terminal will exit after the timeout has expired
off Specifies that the terminal will not exit after the timeout has expired.

Returns
N/A

set end of medium

Description
Defines what will be sent in the end of each medium.

Syntax
set end-of-medium *cr* | *crlf* | *ff*

cr Specifies a carriage return character escape sequence to be sent at the end of the medium.
crlf Specifies a carriage return line feed character escape sequences to be sent at the end of the medium.
ff Stipulates a form feed character escape sequence to be sent at the end of the medium.

Notes
Only for IBM printer emulations.

Returns
N/A

set func cols

Description
Specifies the number of the columns that the user wants to have of function keys.

Syntax
set func-cols *number*

number Specifies the number of columns in the function keys.

Returns
N/A

set func rows

Description
Determines the number of rows for the function buttons.

Syntax
set func-rows *number*

number Specifies the number of rows of function keys.

Returns
N/A

set keyboard

Description
Sets the keyboard type to work with a special VT emulation type.

Syntax
set keyboard *default* | *lk450*

default Specifies the keyboard type to work with the default.
lk450 Specifies the keyboard type to work with lk450.

Notes
Only for VT emulations.

Returns
N/A

Example
```
set keyboard lk450
```

set lu category

Description
Specifies LU category.

Syntax
set lu-category *public* | *pooled* | *dedicated*

public	Specifies *public* as the default value for all LUs.
pooled	Specifies that the SNA session use an LU belonging to a specific LU pool.
dedicated	Specifies that the LU is dedicated to a specific user or device, or use in accessing a particular application for the SNA session.

Notes
Only for 5250 (AS\400) emulations.

All sessions in an LU pool share the same characteristics. When an application program no longer requires the LU, it is returned to the pool to be used by the same or another application program. When you select *pooled*, and an LU pool name is defined, specify the LU pool name in the *pool* field. Otherwise, create an LU pool.

Returns
N/A

set lu name

Description
Specifies the LU name.

Syntax
set lu-name *value*

value	Specifies a string containing the desired LU name.

Notes
Only for 5250 (AS\400) emulations.

Returns
N/A

set max sessions

Description
Specifies the maximum number of PowerTerm sessions that can be opened simultaneously.

Syntax
set max-sessions *number*

number Designates the limit number of PowerTerm sessions that can be opened simultaneously.

Returns
N/A

set menu hotspot rows

Description
Specifies which menu rows will be activated as hotspots.

Syntax
set menu-hotspot-rows *start end*

start Designates the first row of the menu, which will act as a hotspot.
end Designates the last row of the menu, which will act as a hotspot.

Returns
N/A

Example
Causes PowerTerm to display menu hotspots from rows 4 to 20. Except for these lines all other menu fields will not operate as hotspots:

```
set menu-hotspot-rows 4 20
```

set message library

Description
Specifies which library contains the message queue for exception messages.

Syntax

set message-library *libraryname*

libraryname Specifies the location of the library, which contains the message queue for exception messages.

Notes
Only for AS\400 printer sessions.

Returns
N/A

Example
```
set message-library *LIBL
```

set message queue

Description
Designates to which AS/400 message queue, exception messages should be sent.

Syntax
set message-queue *string*

string Represents the message queue where exception messages should be sent.

Notes
Only for AS/400 printer emulations.
A likely scenario: the AS/400 may need to tell the printer to switch to another paper tray.

Returns
N/A

Example
```
set message-queue QSYSOPR
```

set mouse control

Description
Determines whether or not the mouse clicks will be active.

Syntax
set mouse-control *on* | *off*

on Enables mouse clicks.
off Disables mouse clicks.

Returns
N/A

set mouse report

Description
Determines whether or not the mouse clicks will be reported to the host and displayed on the status line or not.
OR
Enables mouse report.
OR
Enables mouse report responses.

Syntax
set mouse-report *on* | *off*

on Enables mouse report.
off Mouse report responses are disabled by default.

Notes
Only for VT emulations.

Returns
N/A

set node name

Description
Specifies the host computer name or the host's IP address through which the data is transferred.

Syntax
set node-name *hostname*

hostname Specifies a string that designates the node name to be used in a CTERM connection type.

Notes
Not relevant to any other connection type.

Returns
N/A

set pad cols

Description
Determines the number of rows for the Power Pad buttons.

Syntax
set pad-cols *number*

number Specifies the columns that the user wants to have in the Power Pad.

Returns
N/A

set pad pos

Description
Determines the x and the y coordinates of the Power Pad window's position.

Syntax
set pad-pos *x y*

x Specifies the upper left corner of the pad in a horizontal screen coordinate.
y Specifies the upper left corner of the pad in a vertical screen coordinate.

Returns
N/A

set pad rows

Description
Determines the number of rows for the Power Pad buttons.

Syntax
set pad-rows *number*

number Specifies the number of the rows in Power Pad.

Returns
N/A

set pad size

Description
Determines the width and height of the Power Pad window.

Syntax
set pad-size *width height*

width	Designates the width of the Power Pad window.
height	Designates the height of the Power Pad window.

Returns
N/A

set parity

Description
Determines the serial communication parity bits and type.

Syntax
set parity [*7* | *8*] [*none* | *even* | *odd* | *mark* | *space*]

7	Designates 7 bits parity as the serial communication type.
8	Designates 8 bits parity as the serial communication type.
none	No parity scheme is designated.
even	Even scheme is designated.
odd	Odd parity scheme is designated.
mark	Mark parity scheme is designated.
space	Space parity scheme is designated.

Notes
Only for Telnet (VT emulation).

Returns
N/A

set print directions

Description
Designates which direction for the printer to print. (Reverses the direction of the sent terminal data to the printer.)

Syntax
set print-direction *r2l | l2r*

r2l Stipulates that the printer will print from right to left.
l2r Stipulates that the printer will print from left to right.

Notes
Only for IBM printer emulations, Hebrew version.

Returns
N/A

set print file name

Description
Sets the print filename.

Syntax
set print-file-name *filename*

filename Stipulates what the name of the file is.

Returns
N/A

set print prefix

Description
Specifies the print prefix.

Syntax
set print-prefix *text*

text Stipulates what the print prefix is.

Returns
N/A

set print screen convert

Description
Sets the slave printer's conversion mode.

Syntax
set print-screen-convert [*no_convert* | *ibm* | *digital* | *graphics*]

no_convert	Indicates that no conversion of data will take place.
ibm	Converts data to IBM character sets.
digital	Converts data to Digital character sets.
graphics	Prints in Graphics mode.

Returns
N/A

set print suffix

Description
Specifies the print suffix.

Syntax
set print-suffix *string*

string	Stipulates what the print suffix is.

Returns
N/A

set printer header lines

Description
Determines the amount of lines of each printer page, which will be allocated for the header.

Syntax
set printer-header-lines *rows*

rows	Represents the amount of lines of each printer page, which will be allocated for the header.

Returns
N/A

set printer transparent header

Description
Determine what to display in the transparent header of each printing.

Syntax
set printer-transparent-header *header*

header Stipulates what will display in the transparent header of each printing.

Returns
N/A

set printer transparent trailer

Description
Determine what to display in the transparent trailer of each printing.

Syntax
set printer-transparent-trailer *trailer*

trailer Stipulates what will display in the transparent trailer of each printing.

Returns
N/A

set protocol type

Description
Determines the flow control parameters.

Syntax
set protocol-type [*none* | *xonxoff* | *hardware*]

none Indicates no flow control at this level or below.
xonxoff Indicates software flow control by recognizing XON and XOFF characters.
hardware Indicates flow control delegated to the lower level, for example a parallel port.

Notes
Valid when working with Telnet (VT emulation), using COM connection.

Returns
N/A

set repeat right alt key

Description
Enables right alt key to repeat when constantly pressed.

Syntax
set repeat-right-alt-key *on* | *off*

on Enables right alt key to repeat when constantly pressed.
off Disables right alt key to repeat when constantly pressed.

Returns
N/A

set repeat left alt key

Description
Allows left alt key to repeat when constantly pressed.

Syntax
set repeat-left-alt-key *on* | *off*

on Enables left alt key to repeat when constantly pressed.
off Disables left alt key to repeat when constantly pressed.

Returns
N/A

set repeat left ctrl key

Description
Allows left ctrl key to repeat when constantly pressed.

Syntax
set repeat-left-ctrl-key *on* | *off*

on Enables left ctrl key to repeat when constantly pressed.
off Disables left ctrl key to repeat when constantly pressed.

Returns
N/A

set repeat right ctrl key

Description
Allows right-ctrl key to repeat when constantly pressed.

Syntax
set repeat-right-ctrl-key *on* | *off*

on Enables right ctrl key to repeat when constantly pressed.
off Disables right ctrl key to repeat when constantly pressed.

Returns
N/A

set security type

Description
Determines the type of security in use.

Syntax
set security-type [*unsecured* | *ssl* | *ssh*]

unsecured Specifies that there is no security implemented in the connection.
ssl Specifies that *SSL* security is implemented in the connection.
ssh Specifies that *SSH* security is implemented in the connection.

Notes
Only for products that support SSL and SSH security types.

Returns
N/A

set slave printer convert

Description
Sets the slave printer's conversion mode.

Syntax
set slave-printer-convert [*no_convert* | *ibm* | *digital* | *graphics*]

no_convert Indicates that no conversion of data will take place
ibm Converts data to IBM character sets.

digital Converts data to Digital character sets.
graphics Prints in Graphics mode.

Returns
N/A

set ssh allow agent

Description
Passes authorization information over the encrypted link.

Syntax
ssh-allow-agent *on* | *off*

on Enables authorization information to be transmitted over the encrypted link.
off Disables authorization information to be transmitted over the encrypted link.

Returns
N/A

set ssh attempt tis

Description
Attempts to authenticate with either TIS or CryptoCard.

Syntax
ssh-attempt-tis *on* | *off*

on Attempts to authenticate with either TIS or CryptoCard.
off No attempts are made to authenticate with either TIS or CryptoCard.

Returns
N/A

set ssh cipher

Description
Specifies the cipher algorithm to be used to encrypt network traffic between the local machine and the server.

Syntax
set ssh-cipher *[3des | blowfish | des | aes]*

The following cipher algorithms are available:

- 3des
- blowfish
- des
- aes

Returns
N/A

set ssh enable compression

Description
Specifies to employ compression.

Syntax
ssh-enable-compression *on | off*

on Enables compression.
off Disables compression.

Returns
N/A

set ssh enable x11

Description
Allows for the X Windows traffic between the X server and X client to be encrypted.

Syntax
set ssh-enable-x11

Returns
N/A

set ssh type

Description
Determines which SSH version to implement.

Syntax
set ssh-type [*ssh1* | *ssh2*]

ssh1 Specifies that the SSH1 type is used.
ssh2 Specifies that the SSH2 type is used.

Notes
The security type must be designated (via script commands or menu) prior to specifying the SSH type.

Returns
N/A

Example
```
set ssh-type ssh1
```

set ssh username

Description
Specifies the user name to be used by the SSH protocol for identification by the host.

Syntax
set ssh-username *sshname*

sshname Represents the user name to be used by the SSH protocol for identification by the host.

Returns
N/A

set ssl type

Description
Determines which SSL version to implement.

Syntax
set ssl-type [*ssl2* | *ssl3* | *tls1*]

ssl2 Specifies that the SSL2 type is used.
ssl3 Specifies that the SSL3 type is used.
tls1 Specifies that the TLS1 type is used.

Notes
The security type must be designated (via script commands or menu) prior to specifying the SSL type.

Returns
N/A

Example
```
set ssl-type ssl3
```

set system name

Description
Specifies the name of the system to which to connect to.

Syntax
set system-name *name*

name Specifies the alphanumeric string which represents the host to which to connect.

Returns
N/A

Example
```
set system-name 126.0.0.200
set system-name as400.eicom.com
```

set telnet port

Description
Specifies the TELNET port number.

Syntax
set telnet-port *value*

value Specifies the number, which represents the desired connection port number.

Notes
Range 1 – 65535. Default port is 23.

Returns
N/A

set terminal id

Description
Specifies the terminal id.

Syntax
set terminal-id *name*

Legal terminal types for *name* are:
- vt52
- vt100
- vt220-7
- vt220-8
- vt320-7
- vt320-8
- vt420-7
- vt420-8
- vt525-7
- vt525-8
- dg
- sco-ansi
- at386
- aixterm
- wyse50
- wyse60
- 3270 display
- 3270 printer
- 5250 display
- 5250 printer
- tvi925
- tvi950
- bbs-ansi
- linux
- siemens
- tandem 6530
- hp 700/96

Returns
N/A

set use alt key up

Description
Determine the behavior of the alt+numeric keypad combination to enter a character using its ASCII code.

Syntax
set use-alt-key-up [*on* | *off*]

on Does not require pressing 3 digits for all entries. This is default.
off Requires pressing 3 digits for all entries.

Returns
N/A

Example
To enter new-line (ASCII code 12):

`set use-alt-key-up` *`on`*

Press simultaneously alt + 1,2 (from the keypad). Release alt to get the desired result.

`set use-alt-key-up` *`off`*

Press simultaneously alt + 1,2 (from the keypad). Release alt to get the same result.

set use available ssh show info

Description
Tells PowerTerm to show the SSH information.

Syntax
set use-available-ssh-show-info [*yes* | *no*]

yes Instructs PowerTerm to show the SSH information.
no Instructs PowerTerm not to show the SSH information.

Returns
N/A

set use tn3270e protocol

Description
Determines whether to implement the TN3270E protocol.

Syntax
set use-tn3270e-protocol [*yes* | *no*]

yes Instructs PowerTerm to use the TN 3270E protocol.
no Instructs PowerTerm not to use the TN 3270E protocol.

Returns
N/A

split

Description
Splits a string into a proper PSL list.

Syntax
split *string* [*splitChars*]

string Specifies the string that splits into a list.
splitChars Specifies the characters that indicate where to split the string.

Notes
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.

Returns
Returns the split string.

Examples
Returns "comp unix misc":

```
split "comp.unix.misc" "."
```

Returns "H e l l o { } w o r l d":

```
split "Hello world" ""
```

start auto print

Description
Starts to accumulate incoming data while it is displayed on the screen.

Syntax

start-auto-print

Returns
N/A

status message

Description
Displays a message in the status bar.

Syntax
status-message *text*

text Specifies the text to appear in the status message.

Returns
An empty string.

Example
```
status-message "The system is temporarily down"
```

stop auto print

Description
Prints all the data accumulated in the printing buffer of the slave printer or in the Auto print buffer.

Syntax
stop-auto-print

Notes
If data was buffered with a printing request and communication failed before the data was sent to the slave printer, execute this command to print the accumulated information.

Returns
N/A

string

Description

Manipulates strings.

Syntax

string *option arg* [*arg ...*]

option Specifies which string operation to perform.
arg Specifies the parameters to be used in the string operation.

Notes

Performs one of several string operations, depending on option. The legal options (which may be abbreviated) are:

string compare *string1 string2*

Performs a character-by-character comparison of strings *string1* and *string2* in the same way as the C *strcmp* procedure.

- Returns -1, 0, or 1, depending on whether *string1* is lexicographically less than, equal to, or greater than *string2*.

string first string1 string2

Searches *string2* for a sequence of characters that exactly match the characters in *string1*.

- If found, returns the index of the first character in the first such match within *string2*.
  If not found, returns -1.

string index *string charIndex*

Returns the *charIndex*'th character of the *string* argument.

- A *charIndex* of 0 corresponds to the first character of the string.
- If *charIndex* is less than 0 or greater than or equal to the length of the string, then an empty string is returned.

string last *string1 string2*

Searches *string2* for a sequence of characters that exactly match the characters in *string1*.

- If found, returns the index of the first character in the last such match within *string2*.
- If there is no match, then returns -1.

string length *string*

Returns a decimal string giving the number of characters in *string*.

string match *pattern string*

If *pattern* matches *string*, returns 1, if it does not, returns 0.

- For the two strings to match, their contents must be identical except that the following special sequences may appear in pattern:
  - ***** Matches any sequence of characters in string, including a null string.
  - **?** Matches any single character in string.
  - **[chars]** Matches any character in the set given by chars. If a sequence of the form x-y appears in chars, then any character between x and y, inclusive, will match.
  - **\x** Matches the single character x. This provides a way of avoiding the special interpretation of the characters *?[]\e in *pattern*.

string range *string first last*
Returns a range of consecutive characters from *string*, starting with the character whose index is *first* and ending with the character whose index is *last*.
- An index of 0 refers to the first character of the string.
- *last* may be *end* to refer to the last character of the string.
- If *first* is less than zero, then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string, then it is treated as if it were *end*.
- If *first* is greater than *last*, then an empty string is returned.

string tolower *string*
Returns a value equal to *string* except that all uppercase letters have been converted to lower case.

string toupper *string*
Returns a value equal to *string* except that all lowercase letters have been converted to upper case.

string trim *string [chars]*
Returns a value equal to *string* except that any leading or trailing characters from the set given by *chars* are removed.
- If *chars* is not specified, then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimleft *string [chars]*
Returns a value equal to *string* except that any leading characters from the set given by *chars* are removed.
- If *chars* is not specified, then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimright *string [chars]*
Returns a value equal to *string* except that any trailing characters from the set given by *chars* are removed.
- If *chars* is not specified, then white space is removed (spaces, tabs, newlines, and carriage returns).

Returns
Returns the converted string.

Example
Returns 9:

```
string last ab 123abc456ab12
```

Returns "UNIX":

```
string toupper Unix
```

switch

Description

Evaluates one of several scripts, depending on a given value.

Syntax

switch [*options*] *string* [*pattern body* [*pattern body* ...]]
or
switch [*options*] *string* {*pattern body* [*pattern body*...]}

options	Specifies which string matching operation to perform.
string	Specifies the string to match to a pattern.
pattern body	Evaluates this argument upon making a match.

Notes

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 PSL 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	Uses exact matching when comparing *string* to *pattern*. This is the default.
-glob	Uses glob-style matching (i.e. the same as implemented by the **string match** command) when matching *string* to the patterns.
-regexp	Uses regular expression matching (i.e. the same as implemented by the **regexp** command) when matching *string* to the patterns.
--	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 multiline switch commands, because 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.

Returns

Returns the output of the last command executed in *body*.

Example

Will return "2":

```
switch abc a-b {format 1} abc {format 2} default {format 3}
```

Will return "1":

```
switch -regexp aaab {
      ^a.*b$      -
      b                 {format 1}
      a*                {format 2}
      default     {format 3}
}
```

Will return "3":

```
switch xyz {
      a                 -
      b                 {format 1}
      a*                {format 2}
      default     {format 3}
}
```

system request

Description

Assigns the terminal to the system request state.

Syntax

system-request [-[*selectionnumber*]]

Notes

Only for 3270/5250 emulations.
The system request is mapped, by default, to the *ALT + F1* key combination for 5250, and *SHIFT + ESC* key combination for 3270.

Returns

N/A

Examples

Opens a dialog where you can manually select the option:

```
send <system-request>
```

Similar to send **<system-request->** followed by manually pressing enter (on AS/400 the system request menu appears):

```
send <system-request->
```

Similar to send **<system-request->** followed by pressing 1 and then enter (on AS/400 option 1 from the system request menu is selected):

```
send <system-request-1>
```

T

T

tell

Description
Returns current access position for an open file.

Syntax
tell *fileId*

fileId Specifies the file upon which the current access position is returned.

Notes
Returns a decimal string giving the current access position in *fileId*.
fileId must have been the return value from a previous call to **open**.

Returns
Returns the file position.

Example
Opens a file and Reads twice 10 bytes. **tell** will return "20":

```
fileId = [open data"]
read $fileId 10
read $fileId 10
position = [tell $fileId]
close $fileId
```

terminal id

Description
Determines the ID returned by the emulation program to the host.

Syntax
terminal-id

Notes

The unprotected field must be followed by another string.

Returns
N/A

time

Description
Measures the execution of a script.

Syntax
time *scriptCommand count*

scriptCommand	Specifies which string operation is being timed.
count	Specifies string, which indicates the average amount of time required per iteration.

Notes
Calls the PSL interpreter *count* times to evaluate *scriptCommand*. If *count* is not specified, it will call it once. It then returns a string of the form "503 microseconds per iteration", which indicates the average amount of time required per iteration, in microseconds.

N*OTE Time is measured in elapsed time, not CPU time.*

Returns
Returns the elapsed time of a PSL command.

Example
Displays the average elapsed time of "**run** test.psl" command:

```
message [time "run test.ppl" 100]
```

toggle-auto-print

Description
Prints simultaneously what is displayed on the screen.

Syntax
toggle-auto-print

Notes
Only for VT emulations.

Returns
N/A

trace

Description
Monitors variable accesses.

Syntax
trace *option* [*arg arg ...*]

option Specifies which method will be employed during the trace.
arg Specifies the parameters to be used during the trace.

Note
This command causes commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented. The legal *option*'s, which may be abbreviated, are:

trace variable *name ops command*

Arranges for *command* to be executed whenever variable *name* is accessed in one of the ways given by *ops*.

name may refer to a normal variable, an element of an array, or to an array as a whole, i.e. *name* may be just the name of an array, with no parenthesized index.

- If *name* refers to a whole array, then *command* is invoked whenever any element of the array is manipulated.
- 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 consists of one or more of the following letters:

- **r** Invokes *command* whenever the variable is read.
- **w** Invokes *command* whenever the variable is written.
- **u** Invokes *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 *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 **r**, **w**, or **u** as defined above.
- *command* executes in the same context as the code that invoked the traced operation. If the variable was accessed as part of a Tcl procedure, then *command* will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *command* invokes a procedure, which it normally does, then the procedure will have to use **upvar** or **uplevel** if it wishes to access the traced variable. Note also that *name1* may not necessarily be the same as the name used to set the trace on the variable. Differences can occur if the access is made through a variable defined with the **upvar** command.

For read and write traces, *command* can modify the variable to affect the result of the traced operation. If *command* modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from *command* is ignored except that if it returns an error of any sort then the traced operation also returns an error with the same error message returned by the trace command. This mechanism can be used to implement read-only variables, for example.

For write traces, *command* is invoked after the variable's value has been changed. It can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, *command* will have to restore the old value of the variable.

While *command* is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by *command* will occur directly, without invoking *command,* or any other traces, again. However, if *command* unsets the variable then unset traces will be invoked. When an unset trace is invoked, the variable has already been deleted. It will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to. The stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked. Any errors in unset traces

are ignored.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the **trace vdelete** command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

- This command returns an empty string.

trace vdelete *name ops command*

If there is a trace set on variable *name* with the operations and command given by *ops* and *command*, then the trace is removed, so that *command* will never again be invoked.

- Returns an empty string.

trace vinfo *name*

Returns a list containing one element for each trace currently set on variable *name*. Each element of the list is itself a list containing two elements, which are the *ops* and *command* associated with the trace.

- If *name* does not exist or does not have any traces set, then the result of the command will be an empty string.

Returns

Returns a string.

U

unlock columns

Description
Ignores any host command that attempts to modify the number of columns in the display area window.

Syntax
unlock-columns

Returns
N/A

unprotected-field

Description
Returns the first unprotected field in a block mode application screen.

Syntax
unprotected-field

Notes
The unprotected field must be followed by another string.

Returns
N/A

Example
```
yd = [unprotected-field 20]
message $yd
```

unset

Description
Deletes variables.

Syntax
unset *name* [*name* ...]

Name Specifies the name of the variable to be deleted.

Notes
This command removes one or more variables.
Each *name* is a variable name, specified in any of the ways acceptable to variable assignment.
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.
An error occurs if any of the variables does not exist, and any variables after the nonexistent one are not deleted.

Returns
Returns an empty string.

Example
Sets and then deletes variable a:

```
a = abc
unset a
```

uplevel

Description
Executes a script in a different stack frame.

Syntax
uplevel [*level*] *arg* [*arg* ...]

level Sets the stack at this specific value.
arg Specifies the parameters to be used by this command.

Notes
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*. If *level* is an integer, then it gives a distance to move (up the procedure calling stack) 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 "#".
uplevel makes it possible to implement new control constructs as PSL procedures. For example, **uplevel** could be used to implement the **while** construct as a PSL procedure.

Returns
Returns the result of the evaluation.

Example
Procedure *a* was invoked from top-level, and it called *b*, and *b* called *c*. *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 {x = 43; d}
```

where d is another PSL procedure. The "x = 43" 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 {x = 42}
```

then the "x = 42" 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.

upvar

Description
Creates links to variables in a different stack frame.

Syntax
upvar [*level*] *otherVar myVar* [*otherVar myVar* ...]

level	Sets the level of the stack at a specific value.
otherVar	Specifies the variable in the stack frame.
myVar	Specifies the variable to be linked to the *OtherVar* in the stack.

Notes
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* is not "#" or a digit (it defaults to 1).
For each *otherVar* argument, **upvar** makes the variable by that name in the procedure frame given by *level* (or at global level, if level is #0 accessible in the current procedure) by the name given in the corresponding *myVar* argument.

The variable named by *otherVar* need not exist at the time of the call. It will be created the first time *myVar* is referenced, just like an ordinary variable.
upvar may only be invoked from within procedures.
myVar may not refer to an element of an array, but *otherVar* may refer to an array element.
The *upvar* command simplifies the implementation of **call-by-name** procedure calling and also makes it easier to build new control constructs as PSL procedures.

Returns
Returns an empty string.

Example
Consider the following procedure:

```
proc add2 name {
 upvar $name x
 x = [expr $x+2]
}
```

add2 is invoked with an argument giving the name of a variable, and it adds two to the value of that variable.
Although *add2* could have been implemented using **uplevel** instead of **upvar**, **upvar** makes it simpler for *add2* to access the variable in the caller's procedure frame.
If an **upvar** variable is **unset** (e.g. x in *add2* above), the **unset** operation affects the variable it is linked to, not the **upvar** variable. There is no way to **unset** an **upvar** variable except by exiting the procedure in which it is defined. However, it is possible to retarget an **upvar** variable by executing another **upvar** command.

use default printer

Description
Sets the Windows default printer in the **Printer Name** field.

Syntax
use–default-printer

Returns
N/A

W

wait

Description

Waits for specific strings received from the host.

Syntax

wait *number for text* [*at row column*]
wait *number seconds*

number	Specifies the number of seconds required to wait.
text	Specifies the text that is expected.
row column	Specifies where to expect *text* to appear.

Notes

This command instructs PowerTerm to wait a specified period of time for a certain string to arrive from the host application. This text is case sensitive. If the string does not arrive from the host application within the specified time, PowerTerm returns the appropriate value.

wait system

Only for 5250 emulations.
It instructs PowerTerm to wait for the AS/400 to notify it when it is finished processing a screen. At the time of processing the screen data the emulator displays *X SYSTEM* in the status bar and the user cannot enter any commands. When the AS/400 finishes processing, the X SYSTEM disappears and the script continues with the next command.
If the **wait system** script command is executed when the X SYSTEM is not displayed, it will immediately continue with the next script command.

wait record

Only for 3270 emulations.
It instructs PowerTerm to wait for the next screen record from the mainframe. When the mainframe finishes processing the screen record, the script continues with the next command.
If the **wait record** script command is executed when the mainframe is not processing a screen record, it will immediately continue with the next script command.
The following note relates to wait system and wait record**.**

Between different screens in the AS/400 environment, there will only be one X SYSTEM. However, in the mainframe environment, there may be several records per screen. For examples on the use of **wait system** and **wait record** commands, select the **Script | Start Script Recording** menu command. Enter several screens into the application and select the **Script | Stop Script Recording** menu command. You can then save the script and view it.

Examples

The following script commands instruct PowerTerm to wait for two strings, "username" and "password", during the login process. If the string is not received within 10 seconds, the message “username not found” or “password not found”, displays and the script terminates.

```
Found = [wait 10 for username]
if {$found == 0} {message "Username not found";return}
send "john^M"
Found = [wait 10 for password]
if {$found == 0} {message "Password not found";return}
send "john pass^M"
wait 10 seconds
```

wait string

Description

Waits for multiple strings.

Syntax

wait string *stringtosearchfor*

stringtosearchfor Specifies the string to search for.

Notes

Every time you enter a **wait string** command, the string is inserted into a list of strings. The limit of the total number of **wait string** commands is 8, and the length of each string cannot exceed 50 characters.

In order for the search to begin, the command **wait start** must be executed. When you execute the command, PowerTerm starts to wait for one of these designated strings, via the **wait string** command.

When the **wait start** command finds one of the listed strings, it returns an integer value and releases the script to continue running. The return value represents the number of the string according to its initial order. Every time that a **wait start** command has successfully located a string, the strings’ list is emptied and has to be reentered.

You cannot run an additional **wait start** while one is already running and is waiting for strings.

Returns

N/A

Example

This script enters into the string list the strings *David, wants, to,* and *sleep*:

```
wait string David
wait string wants
wait string to
wait string sleep
```

The **wait start** command will cause PowerTerm script to wait:

```
x = [wait start]
message x
```

If the string *David* is received then the message displaying the content of the variable *x* will show "1". However if the string *sleep* is received then a message saying "4" will be displayed.

while

Description

Executes script repeatedly as long as a condition is met.

Syntax

while *test body*

test	Specifies the condition, which must be evaluated.
body	Specifies what will be executed if *test* is True.

Notes

The **while** command evaluates *test* as an expression, in the same way that the **expr** command evaluates its argument.
The value of the expression must be a proper boolean value. If it is True, then **body** is executed by passing it to the PSL.
After *body* is executed, *test* is evaluated again, and the process repeats until eventually *test* evaluates to False. **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.

Returns

Returns an empty string.

Example

Displays "yes" while variable host is equal to "vax":

```
while {$host == vax} {
      message yes
      commands...
```

}

window

Description
Changes the emulation window status.

Syntax
window *option* [*args*...]

option	Specifies the condition to be evaluated.
args	Specifies what is executed if *test* is True.

Notes
Performs one of several window operations, depending on option. The legal options are:

window [*maximize* | *minimize* | *restore* | *hide* | *show*]	Changes the emulation window accordingly.
window *size height width*	Changes the height and width (in pixels) of the window.
window *position y x*	Changes the x- and y-coordinates (in pixels) of the window.

Returns
Returns an empty string.

Example
Restores the window and sets its position and size:

```
window restore
window position 50 50
window size 400 600
```