PATROL® Script Language Reference Manual Volume 2—PSL Functions

Version 3.5

January 15, 2003

bmcsoftware

Copyright 1994–2002 BMC Software, Inc., as an unpublished work. All rights reserved.

BMC Software, the BMC Software logos, and all other BMC Software product or service names are registered trademarks or trademarks of BMC Software, Inc. All other registered trademarks or trademarks belong to their respective companies.

THE USE AND CONTENTS OF THIS DOCUMENTATION ARE GOVERNED BY THE SOFTWARE LICENSE AGREEMENT ENCLOSED AT THE BACK OF THIS DOCUMENTATION.

Restricted Rights Legend

U.S. GOVERNMENT RESTRICTED RIGHTS. UNPUBLISHED–RIGHTS RESERVED UNDER THE COPYRIGHT LAWS OF THE UNITED STATES. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in FAR Section 52.227-14 Alt. III (g)(3), FAR Section 52.227-19, DFARS 252.227-7014 (b), or DFARS 227.7202, as amended from time to time. Contractor/Manufacturer is BMC Software, Inc., 2101 CityWest Blvd., Houston, TX 77042-2827, USA. Any contract notices should be sent to this address.

Contacting BMC Software

You can access the BMC Software Web site at **http://www.bmc.com**. From this Web site, you can obtain information about the company, its products, corporate offices, special events, and career opportunities.

United States and Canada

Address BMC Software, Inc.
2101 CityWest Blvd.
Houston TX 77042-2827

Telephone 713 918 8800 or
800 841 2031

Fax 713 918 8000

Outside United States and Canada

Telephone (01) 713 918 8800

Fax (01) 713 918 8000

Customer Support

You can obtain technical support by using the Support page on the BMC Software Web site or by contacting Customer Support by telephone or e-mail. To expedite your inquiry, please see “Before Contacting BMC Software.”

Support Web Site

You can obtain technical support from BMC Software 24 hours a day, 7 days a week at **http://www.bmc.com/support.html**. From this Web site, you can

- read overviews about support services and programs that BMC Software offers
- find the most current information about BMC Software products
- search a database for problems similar to yours and possible solutions
- order or download product documentation
- report a problem or ask a question
- subscribe to receive e-mail notices when new product versions are released
- find worldwide BMC Software support center locations and contact information, including e-mail addresses, fax numbers, and telephone numbers

Support by Telephone or E-mail

In the United States and Canada, if you need technical support and do not have access to the Web, call 800 537 1813. Outside the United States and Canada, please contact your local support center for assistance. To find telephone and e-mail contact information for the BMC Software support center that services your location, refer to the Contact Customer Support section of the Support page on the BMC Software Web site at **www.bmc.com/support.html**.

Before Contacting BMC Software

Before you contact BMC Software, have the following information available so that Customer Support can begin working on your problem immediately:

- product information
  - product name
  - product version (release number)
  - license number and password (trial or permanent)
- operating system and environment information
  - machine type
  - operating system type, version, and service pack or other maintenance level such as PUT or PTF
  - system hardware configuration
  - serial numbers
  - related software (database, application, and communication) including type, version, and service pack or maintenance level

- sequence of events leading to the problem
- commands and options that you used
- messages received (and the time and date that you received them)
  - product error messages
  - messages from the operating system, such as `file system full`
  - messages from related software

BMC Software, Inc., Confidential and Proprietary Information

Contents

About This Manual xv

Chapter 1 PSL Built-in Functions

acos() 1-9
add_diary() 1-11
annotate() 1-13
Specifying the annotate() Function Output Format 1-16
annotate_get() 1-19
asctime() 1-22
asin() 1-26
atan() 1-27
atexit() 1-28
batch_set 1-30
blackout() 1-33
cat() 1-38
ceil() 1-40
chan_exists() 1-41
change_state() 1-42
Rule State and Combined State 1-43
chart() 1-45
Graph Definition Format (GDF) Syntax 1-47
close() 1-54
code_cvt() 1-56
cond_signal() 1-59
cond_wait() 1-60
console_type() 1-64
convert_base() 1-65
convert_date() 1-68

BMC Software, Inc., Confidential and Proprietary Information

convert_locale_date() 1-74
cos() 1-78
cosh() 1-81
create() 1-83
date() 1-88
dcget_text() 1-89
debugger() 1-91
desktop() 1-92
destroy() 1-93
destroy_lock() 1-94
dget_text() 1-95
difference() 1-97
dump_hist() 1-99
encrypt() 1-103
event_archive() 1-105
event_catalog_get() 1-111
event_check() 1-114
event_query() 1-119
event_range_manage() 1-127
event_range_query() 1-129
event_report() 1-132
Output 1-136
event_schedule() 1-138
event_trigger() 1-146
event_trigger2() 1-149
execute() 1-152
exists() 1-156
exp() 1-158
fabs() 1-159
file() 1-160
floor() 1-164
fmod() 1-165
fopen() 1-166
fopen() Function and Named Pipes 1-169
fseek() 1-171
ftell() 1-174
full_discovery() 1-176
get() 1-177
get_chan_info() 1-178
get_ranges() 1-182

BMC Software, Inc., Confidential and Proprietary Information

get_text() 1-186
get_vars() 1-188
getenv() 1-190
getpid() 1-192
getpname() 1-193
grep() 1-194
history() 1-198
history_get_retention() 1-201
index() 1-202
int() 1-203
internal() 1-204
intersection() 1-205
in_transition() 1-206
isnumber() 1-208
is_var() 1-210
join() 1-212
kill() 1-213
length() 1-214
lines() 1-215
lock() 1-216
log() 1-219
loge() 1-220
log10() 1-222
matchline() 1-224
msg_check() 1-227
msg_get_format() 1-229
msg_get_severity() 1-231
msg_printf() 1-233
msg_sprintf() 1-235
ntharg() 1-237
nthargf() 1-240
nthline() 1-243
nthlinef() 1-246
num_bytes() 1-248
num_consoles() 1-251
pconfig() 1-253
pconfig() Function Messages and Return Codes 1-260
popen() 1-265
poplines() 1-267
popup_report() 1-269

BMC Software, Inc., Confidential and Proprietary Information

pow() 1-272
print() 1-274
printf() 1-275
proc_exists() 1-280
process() 1-281
PslExecute() 1-284
PslSetOptions() 1-287
random() 1-288
read() 1-290
readln() 1-294
refresh_parameters() 1-296
remote_close() 1-298
remote_event_query() 1-299
Specifying the remote_event_query() Output Format 1-304
remote_event_trigger() 1-306
remote_file_send() 1-309
remote_open() 1-315
Using the Remote File Functions 1-317
remove() 1-323
replace() 1-325
response() 1-326
response() Function Preferences 1-339
response() Function Elements 1-351
response_get_value() 1-401
rindex() 1-404
sec_store_get() 1-405
sec_store_set() 1-407
set() 1-410
set_alarm_ranges() 1-412
set_locale() 1-416
share() 1-419
sin() 1-421
sinh() 1-424
sleep() 1-427
snmp_agent_config() 1-429
snmp_agent_start() 1-431
snmp_agent_stop() 1-433
snmp_close() 1-434
snmp_config() 1-435
_snmp_debug() 1-440

BMC Software, Inc., Confidential and Proprietary Information

snmp_get() 1-441
snmp_get_next() 1-444
snmp_h_get() 1-447
snmp_h_get_next() 1-450
snmp_h_set() 1-453
snmp_open() 1-456
snmp_set() 1-459
snmp_trap_ignore() 1-462
snmp_trap_listen() 1-463
snmp_trap_raise_std_trap() 1-466
snmp_trap_receive() 1-467
snmp_trap_register_im() 1-472
snmp_trap_send() 1-476
snmp_walk() 1-479
sort() 1-481
splitline() 1-484
sprintf() 1-487
sqrt() 1-493
srandom() 1-495
subset() 1-496
substr() 1-498
system() 1-499
tail() 1-501
tan() 1-503
tanh() 1-505
text_domain() 1-507
time() 1-509
tmpnam() 1-510
tolower() 1-511
toupper() 1-512
trace_psl_process() 1-513
trim() 1-515
union() 1-518
unique() 1-520
unlock() 1-521
unset() 1-523
va_arg() 1-524
va_start() 1-526
write() 1-528
write_to_report() 1-530

BMC Software, Inc., Confidential and Proprietary Information

Chapter 2 **PSL Perform Agent Metric Functions**

Introduction 2-2
Metrics and Metric Groups 2-2
Metric Data Types and Computations 2-3
closeMetric() 2-7
closeMetricGroup() 2-10
disableMetric() 2-12
enableMetric() 2-14
enableMetricGroup() 2-16
findRowNumber() 2-18
getMetricComputation() 2-20
getMetricValue() 2-23
getRowCount() 2-26
openMetric() 2-28
openMetricGroup() 2-31
queryMetricComputations() 2-33
queryMetricGroups() 2-36
queryMetrics() 2-38
setMetricComputation() 2-40
updateMetricGroup() 2-43

Index

BMC Software, Inc., Confidential and Proprietary Information

Figures

Figure 1-1 Plain-Text Annotation Dialog Box 1-16
Figure 1-2 InfoBox Annotation Dialog Box 1-17
Figure 1-3 response() Function Annotation Dialog Box 1-18
Figure 1-4 Default response() Function Dialog Box 1-329
Figure 1-5 Text Field Example 1-357
Figure 1-6 Radio Button Example 1-359
Figure 1-7 Check Box Example 1-361
Figure 1-8 Toggle Button Example 1-364
Figure 1-9 Selection List Example 1-366
Figure 1-10 Option Menu Example 1-369
Figure 1-11 Clicker Example 1-371
Figure 1-12 Time Spinner Example 1-373
Figure 1-13 Sliding Scale Example 1-375
Figure 1-14 Label Example 1-377
Figure 1-15 Icon Example 1-378
Figure 1-16 Separator Example 1-380
Figure 1-17 Frame Example 1-382
Figure 1-18 Column Example 1-384
Figure 1-19 Row Example 1-386
Figure 1-20 Popup Example 1-388
Figure 1-21 response() Function Example Dialog Box 1-398
Figure 2-1 perform agent Metric Illustration 2-3

BMC Software, Inc., Confidential and Proprietary Information

BMC Software, Inc., Confidential and Proprietary Information

Tables

Table 1-1 PSL Built-in Functions 1-1
Table 1-2 Format Symbols for asctime() 1-23
Table 1-3 Return Values for batch_set(). 1-31
Table 1-4 Supported Codeset Names 1-57
Table 1-5 Format Symbols for convert_locale_date() 1-75
Table 1-6 event_catalog_get() Function Macro Variables 1-112
Table 1-7 event_query() Function Macro Variables. 1-124
Table 1-8 Format of the Result of the get_chan_info() Function. 1-179
Table 1-9 Valid Values for the Result of the get_chan_info() function 1-179
Table 1-10 Decimal and Octal File Permissions 1-313
Table 1-11 Return Values for response() Function Elements. 1-332
Table 1-12 Elements of a Dynamic response() Function 1-337
Table 1-13 response() Function Preferences 1-341
Table 1-14 Simple Control Elements 1-352
Table 1-15 Compound Control Elements. 1-353
Table 1-16 response() Function Elements 1-356
Table 2-1 Metric Data Types 2-4
Table 2-2 Metric Computations and Supported Data Types 2-5
Table 2-3 Metric Computation Equations 2-5

BMC Software, Inc., Confidential and Proprietary Information

BMC Software, Inc., Confidential and Proprietary Information

About This Manual

This book presents the PATROL® Script Language (PSL), a comprehensive language for writing complex application discovery procedures, parameters, and commands within the PATROL monitoring environment. The book you are reading, *PATROL Script Language Reference Manual Volume 2—PSL Functions*, is the second volume of the PSL documentation, and it presents detailed information about the built-in functions. Refer to the first volume of this series if you are unfamiliar with the basics of PSL development.

Terms of Usage

The recipient of this document acknowledges and agrees that the PATROL Script Language (PSL) is proprietary and confidential to BMC Software, Inc., is only for the internal use of the intended recipient, and is only for use as described herein. The recipient further agrees that this document shall not be copied, disclosed, or transferred to any third party without the prior, express, written consent of BMC Software. Any use of this documentation or the PATROL software product constitutes acceptance of these terms. If these terms are not acceptable, promptly return this documentation or the PATROL software product to BMC Software.

Who Should Read This Book

The *PATROL Script Language Reference Manual* is intended for advanced users of PATROL who need to customize and extend the PATROL® Agent, PATROL® Console, and Knowledge Module® (KM) monitoring environment. It assumes you are familiar with a third- or fourth-generation programming language such as C, Perl, TCL, or a Unix programming shell. This volume presents detailed information about the built-in functions, and you should use it if you are already familiar PSL basics such as statement syntax and data types. Otherwise, refer to the *PATROL Script Language Reference Manual Volume 1—PSL Essentials*.

How This Manual Is Organized

The *PATROL Script Language Reference Manual Volume 2—PSL Functions* is organized into the following chapters.

Chapter	Title	Purpose
1	"PSL Built-in Functions"	Describes the PSL built-in functions. This chapter alphabetically lists almost all of the functions.
2	"PSL Perform Agent Metric Functions"	Describes the PSL built-in functions that were specifically designed for the Perform Agent. These functions are presented separately because they work only on Windows platforms.

Related Documentation

BMC Software products offer several types of documentation:

- online and printed books
- online Help
- release notes

BMC Software, Inc., Confidential and Proprietary Information

Online and Printed Books

The books that accompany BMC Software products are available in online format and printed format. You can view online books with Acrobat Reader from Adobe Systems. The reader is provided at no cost, as explained in “To Access Online Books.” You can also obtain additional printed books from BMC Software, as explained in “To Request Additional Printed Books.”

To Access Online Books

Online books are formatted as Portable Document Format (PDF) files. You can view them, print them, or copy them to your computer by using Acrobat Reader 3.0 or later.You can access online books from the documentation compact disc (CD) that accompanies your product or from the World Wide Web.

In some cases, installation of Acrobat Reader and downloading the online books is an optional part of the product-installation process. For information about downloading the free reader from the Web, go to the Adobe Systems site at **http://www.adobe.com**.

To view any online book that BMC Software offers, visit the support page of the BMC Software Web site at **http://www.bmc.com/support.html**. Log on and select a product to access the related documentation. (To log on, first-time users can request a user name and password by registering at the support page or by contacting a BMC Software sales representative.)

To Request Additional Printed Books

BMC Software provides a core set of printed books with your product order. To request additional books, go to **http://www.bmc.com/support.html**.

BMC Software, Inc., Confidential and Proprietary Information

Online Help

You can access Help for a product through the product's Help menu. The online Help provides information about the product's graphical user interface (GUI) and provides instructions for completing tasks.

Release Notes

Printed release notes accompany each BMC Software product. Release notes provide up-to-date information such as

- updates to the installation instructions
- last-minute product information

The latest versions of the release notes are also available on the Web at **http://www.bmc.com/support**.

Conventions

This *PATROL Script Language Reference Manual* uses several format and font conventions to make it more usable and to make it more compatible with other computer programming language reference manuals. The following conventions are used in this book:

BMC Software, Inc., Confidential and Proprietary Information

- This book includes special elements called *notes*, *warnings*, *examples*, and *tips*:

Note

Notes provide additional information about the current subject.

Warning

Warnings alert you to situations that can cause problems, such as loss of data, if you do not follow instructions carefully.

Example

An example clarifies a concept discussed in text.

Tip

A tip provides useful information that may improve product performance or make procedures easier to follow.

- `All syntax, operating system terms, and literal examples are presented in this typeface.`

- In instructions, **boldface** type highlights information that you enter. File names, directories, and Web addresses also appear in boldface type.

- The symbol => connects items in a menu sequence. For example, **Actions** => **Create Test** instructs you to choose the Create Test command from the Actions menu.

- The symbol » denotes one-step instructions.

BMC Software, Inc., Confidential and Proprietary Information

- Path names, or system messages, *italic* text represents a variable, as shown in the following examples:

  `The table` *`table_name`* `is not available.`

  system/*instance/file_name*

- A vertical bar (|) separating items indicates that you must choose one item. In the following example, you would choose *a*, *b*, or *c*:

  `a | b | c`

- An ellipsis (...) indicates that you can repeat the preceding item or items as many times as necessary. The PSL statement and built-in function descriptions use ellipses (...) to indicate parameters that may be indefinitely replicated within the statement or function. For example, the following function format indicates that *variable* can be specified an arbitrary number of times within the `snmp_get()` function:

  snmp_get(*session*,*variable1*,[...,*variablen*])

  The following statement format indicates that the construction `case` *n*: {*BLOCK*} can be repeated within the statement an arbitrary number of times.

  switch (*variable*)
  {
  case *a*: {*BLOCK*}
  case *b*: {*BLOCK*}
  . . .
  case *n*: {*BLOCK*}
  default: {*BLOCK*}
  }

- Square brackets ([ ]) around an item indicate that the item is optional.

BMC Software, Inc., Confidential and Proprietary Information

Statement and Built-in Function Descriptions

The PSL statement and built-in function descriptions in Chapters 3 and 4 use standard programming language reference formatting. That format consists of the following conventions:

- statement or function title and an optional single-sentence description
- a detailed format and parameter description
- a detailed description of statement or function action, restrictions, dependencies, and error conditions
- statement or built-in function descriptions begin a new page

Mouse Controls

The following table shows equivalent mouse buttons for Unix users and Windows users:

Unix Button	Windows Button	Description
MB1	left mouse button	Click this button on an icon or menu command to select that icon or command. Click MB1 on a command button to initiate action. Double-click an icon to open its container.
MB2	not applicable	Click this button on an icon to display the InfoBox for the icon. To simulate MB2 on a two-button mouse, simultaneously press the two buttons (MB1 and MB3).
MB3	right mouse button	Click this button on an icon to display its pop-up menu.

Note

If you have a one-button mouse (such as an Apple Macintosh mouse), assign MB1 to that button. Also define a user-selectable combination of option and arrow keys to simulate MB2 and MB3. For details, refer to the documentation for your emulation software.

BMC Software, Inc., Confidential and Proprietary Information

PSL Built-in Functions

This chapter describes the PSL built-in functions. The functions are listed in alphabetical order in this chapter. Table 1-1 lists the PSL built-in functions described in this manual, indexed by category:

Table 1-1 PSL Built-in Functions (Part 1 of 8)

Function	Definition	Page
Application Discovery:		
full_discovery()	verify whether the PSL script is in a full discovery cycle	1-176
in_transition()	allow time for an object to change states	1-206
proc_exists()	verify the existence of a process on the system	1-280
process()	return a list of processes in the PATROL Agent process cache	1-281
Command Execution:		
chan_exists()	verify that a process or file I/O channel exists	1-41
close()	close a channel created by the popen function	1-54
desktop()	execute a PATROL link command within the Console	1-92
execute()	submit command to a command processor	1-152
fopen()	open a channel to a file	1-166
get_chan_info()	return channel information	1-178
internal()	execute command within the PATROL Agent	1-204
popen()	open a channel to a process	1-265
share()	convert a local channel to a shared global channel	1-419
system()	submit a command to the operating system	1-499
Condition Variable:		
cond_signal()	send a condition variable to unblock a process	1-59
cond_wait()	block a process while waiting for a condition variable	1-60

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 2 of 8)

Function	Definition	Page
Date and Time:		
asctime()	return the time/date as a character string	1-22
convert_date()	convert the Gregorian date and time to system time	1-68
date()	return the current date and time	1-88
time()	return the current system time	1-509
Debugging:		
debugger()	suspend process and wait for “attach” from the PSL Debugger	1-91
Environment:		
getenv()	return the value of an object environment variable	1-190
Event Management:		
add_diary()	write text to the diary of a PATROL event	1-11
event_archive()	write selected PATROL events to an archive file	1-105
event_catalog_get()	return information from a PATROL event catalog	1-111
event_check()	count specific PATROL Event Manager events	1-114
event_query()	return specific PATROL Event Manager events	1-119
event_range_manage()	change the state of a range of event IDs	1-127
event_range_query()	return the events with a specified range of event IDs	1-129
event_report()	return statistics for events matching the specified filter	1-132
event_schedule()	schedule a PATROL event	1-138
event_trigger()	create an event instance	1-146
event_trigger2()	create an event instance with a specified origin	1-149
File Handling:		
cat()	return the contents of a file as a single string	1-38
file()	test for file existence and return the last modification time of the file	1-160
fseek()	set file position	1-171
ftell()	return current file position	1-174
remove()	remove a file from the file system	1-323

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 3 of 8)

Function	Definition	Page
Input/Output:		
chart()	load, destroy, or print a chart on the PATROL Console	1-45
log()	send an information event to the PATROL Event Manager	1-219
popup_report()	pop up a task window on interested Consoles	1-269
print()	return format string to a PATROL Console computer system window	1-274
printf()	display output using C language formatting conventions	1-275
read()	return data read from an open channel	1-290
readln()	return a line of data read from an open channel	1-294
sprintf()	return format string to the specified destination	1-487
tmpnam()	return a unique temporary file name character string	1-510
write()	write data to an open channel	1-528
write_to_report()	write arbitrary text to a pop-up task window	1-530
International:		
code_cvt()	convert a string from one codeset to another	1-56
convert_locale_date()	change the format of a date string	1-74
num_bytes()	return the length of a string in bytes	1-248
set_locale()	set or get the value of a locale category	1-416
International (ID-Based Messaging):		
msg_check()	validate a message ID and message catalog	1-227
msg_get_format()	return the format information for a message	1-229
msg_get_severity()	return a code that explains the severity of a message	1-231
msg_printf()	display message in the computer or task output window	1-233
msg_sprintf()	return message text	1-235
International (String-Based Messaging):		
dcget_text()	return message text by defining domain and category	1-89
dget_text()	return message text by defining domain	1-95
get_text()	return message text	1-186
text_domain()	set or get the current name of the domain	1-507
Locking:		
destroy_lock()	destroy a PSL process lock and wake blocked processes	1-94
lock()	request a PSL process lock	1-216
unlock()	release a PSL process lock	1-521

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 4 of 8)

Function	Definition	Page
Mathematical:		
acos()	return the arccosine trigonometric function	1-9
asin()	return the arcsine trigonometric function	1-26
atan()	return the arctangent trigonometric function	1-27
ceil()	return the smallest integer not less than the argument	1-40
convert_base()	convert a number from one base to another	1-65
cos()	return the cosine trigonometric function	1-78
cosh()	return the hyperbolic cosine function	1-81
exp()	return the natural base *e* raised to a power	1-158
fabs()	return the absolute value of a floating point number	1-159
floor()	return the largest integer not greater than the argument	1-164
fmod()	return the remainder of a floating point division	1-165
int()	return the integer portion of a number	1-203
loge()	return the logarithm with respect to base *e*	1-220
log10()	return the logarithm with respect to base 10	1-222
pow()	return the value of a number raised to a power	1-272
random()	return a pseudorandom number	1-288
sin()	return the sine trigonometric function	1-421
sinh()	return the hyperbolic sine function	1-424
sqrt()	return the square root	1-493
srandom()	seed the pseudorandom number generator	1-495
tan()	return the tangent trigonometric function	1-503
tanh()	return the hyperbolic tangent function	1-505
Object Manipulation:		
batch_set()	similar to set() except many variables are set at one time	1-30
blackout()	hide PATROL object state changes for a period of time	1-33
change_state()	change the state of an object	1-42
create()	add a new application instance	1-83
destroy()	delete an existing application instance	1-93
exists()	verify whether the object exists	1-156
get()	return the current value of a variable	1-177
get_vars()	list the variables associated with an object	1-188
is_var()	return whether an object is a variable	1-210
set()	specify the value of a variable	1-410
unset()	remove an object assigned using the set() function	1-523

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 5 of 8)

Function	Definition	Page
Parameters and Parameter History:		
annotate()	provide contextual information for a PATROL parameter	1-13
annotate_get()	return the annotation for a PATROL parameter value	1-19
dump_hist()	dump the history for applications, instances, and parameters	1-99
get_ranges()	return the Border, Alarm1, and Alarm2 range settings	1-182
history()	return event history information for an object parameter	1-198
history_get_retention()	return the history retention value for a PATROL object	1-201
refresh_parameters()	refresh PATROL parameters	1-296
set_alarm_ranges()	set the range for the alarm conditions for a parameter	1-412
PATROL Agent:		
pconfig()	manage PATROL Agent configuration variables	1-253
PATROL Console:		
console_type()	return 1 if the Console is a PATROL Developer Console	1-64
num_consoles()	return the number of consoles accessing this script	1-251
PATROL Console User Input:		
response()	display a dialog box to the PATROL Console	1-326
response_get_value()	return element status from a dynamic response() function	1-401
Perform Agent Data Access (Windows Only): These functions are discussed in Chapter 2, “PSL Perform Agent Metric Functions.”		
closeMetric()	close a handle to a metric	2-7
closeMetricGroup	close a handle to a metric group	2-10
disableMetric()	discontinue data collection for a given metric	2-12
enableMetric()	enable a metric for collection	2-14
enableMetricGroup()	enable a metric group for data collection	2-16
findRowNumber()	find the row numbers that contains a given value	2-18
getMetricComputation()	get the current metric computation	2-20
getMetricValue()	return the data value for a metric on a given row	2-23
getRowCount()	get the number of rows of data for the given data handle	2-26
openMetric()	open a handle to a metric	2-28
openMetricGroup()	open a handle to a metric group	2-31
queryMetricComputations()	get a list of available computations for a metric	2-33
queryMetricGroups()	return a list of available metric groups	2-36
queryMetrics()	return a list of available metrics in the given metric group	2-38
setMetricComputation()	set the computation for a metric	2-40
updateMetricGroup()	refresh the data for a metric group	2-43

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 6 of 8)

Function	Definition	Page
Process Control:		
atexit()	call a user-defined PSL function upon the normal termination of a process	1-28
getpid()	return the process identifier for the current PSL process	1-192
getpname()	return a PSL process name as per a specified process ID	1-193
kill()	terminate a PSL process	1-213
PslExecute()	start a PSL process	1-284
PslSetOptions()	set the run-time options for the calling PSL process	1-287
sleep()	suspend PSL process execution for a specified number of seconds	1-427
trace_psl_process()	enable tracing for a specified process, application, or parameter	1-513
Remote PATROL Agent:		
remote_close()	close a session with a remote PATROL Agent	1-298
remote_event_query()	return a list of events from a remote Event Manager	1-299
remote_event_trigger()	create an event instance for a remote Event Manager	1-306
remote_file_send()	send a file to a remote PATROL Agent	1-309
remote_open()	open a session with a remote PATROL Agent	1-315
Security:		
sec_store_get()	get value from the secure area of a PATROL Agent	1-405
sec_store_set()	store and encrypt a text string in the PATROL Agent	1-407
Set:		
difference()	return a list whose elements are in one list but not others	1-97
intersection()	return a list whose elements are common to all specified lists	1-205
sort()	sort a list of characters or numbers	1-481
subset()	verify that one list is a a subset of another	1-496
union()	return a list that is the union of individual lists	1-518
unique()	remove all duplicate elements from a list	1-520

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 7 of 8)

Function	Definition	Page
SNMP:		
snmp_agent_config()	verify PATROL Agent SNMP support	1-429
snmp_agent_start()	start the PATROL Agent SNMP support	1-431
snmp_agent_stop()	stop the PATROL Agent SNMP support	1-433
snmp_close()	close a PATROL Agent SNMP session with the PATROL Console	1-434
snmp_config()	configure or return parameters from a PATROL SNMP session	1-435
_snmp_debug()	activate debugging for SNMP	1-440
snmp_get()	return variables from the Management Information Base (MIB)	1-441
snmp_get_next()	return alphabetically greater variables from the MIB	1-444
snmp_h_get()	open a brief session to return variables from the MIB	1-447
snmp_h_get_next()	open a brief session to return alphabetically greater variables from the MIB	1-450
snmp_h_set()	open a brief session to set variables in the MIB	1-453
snmp_open()	open a session with the SNMP agent	1-456
snmp_set()	set variables in the MIB	1-459
snmp_trap_ignore()	stop accumulating incoming SNMP event traps	1-479
snmp_trap_listen()	start accumulating incoming SNMP event traps	1-453
snmp_trap_raise_std_trap()	raise the standard PATROL trap	1-456
snmp_trap_receive()	receive and return SNMP event traps	1-459
snmp_trap_register_im()	maintain a list of SNMP Managers that receive PATROL SNMP traps	1-472
snmp_trap_send()	send an SNMP trap	1-462
snmp_walk()	return subsequent variables from the MIB	1-463

BMC Software, Inc., Confidential and Proprietary Information

Table 1-1 PSL Built-in Functions (Part 8 of 8)

Function	Definition	Page
String Manipulation:		
encrypt()	encrypt a supplied string as per a specified encryption type	1-103
grep()	list all lines that contain a specified pattern	1-194
index()	return the character offset of one string within another	1-202
isnumber()	determine whether the string is a valid numeric expression	1-208
join()	join strings together with a separator between each string	1-212
length()	return the length of a text string or block	1-214
lines()	return the number of lines in a text string or block	1-215
matchline()	save the results of a regular-expression to variables	1-224
ntharg()	return the specified fields from a text string or block	1-237
nthargf()	return the specified fields from a text string or block (multiple adjacent delimiters delimit NULL fields)	1-240
nthline()	return the specified lines from a text string or block (skip new-line characters)	1-243
nthlinef()	return the specified lines from a text string or block	1-246
poplines()	return the specified lines from a variable	1-267
replace()	replace a specified string in text with a replacement string	1-325
rindex()	return the position of the last occurrence of one string within another	1-404
splitline()	parse text into user-defined variables	1-484
substr()	return a specified portion of a string	1-498
tail()	return the last lines from a text string or block	1-501
tolower()	return a string with all characters converted to lowercase	1-511
toupper()	return a string with all characters converted to uppercase	1-512
trim()	return a string with specified characters removed	1-515
Variable Argument List:		
va_arg()	return the next value from a variable argument list	1-524
va_start()	reset a variable argument list to return the first value	1-526

BMC Software, Inc., Confidential and Proprietary Information

acos()

Return the arccosine of the argument

Format

```
acos(cosine)
```

Parameter

Parameter	Definition
`cosine`	cosine argument **Valid Values** $-1 \le cosine \le 1$

Description

The acos() function returns the arccosine of *cosine*, that is, the length in radians of the arc whose cosine is *cosine*. The output range for the acos() function is $0 \le \text{acos}() \le \pi$. The `acos()` function return value is accurate to six decimal places.

Example

The following PSL statements test the roundoff error in the cosine and arccosine functions by comparing argument with acos(cos(*argument*)) for $0 \leq argument \leq 3.1$ ($\approx\pi$) radians in increments of 0.1 radian.

```
arg = 0.0;
printf ("radians\t\tcos()\t\tacos(cos())\troundoff error\n");
printf ("-------\t\t-----\t\t-----------\t--------------\n");
while (arg <= 3.1) {
   cosine = cos(arg);
   argument = acos(cosine);
   error = argument - arg;
   printf ("%+1.1f\t\t%+1.6f\t%+1.6f\t%+1.6f\n",arg,cosine,argument,error);
   arg = arg + 0.1;
}
```

BMC Software, Inc., Confidential and Proprietary Information

add_diary()

Add text to the diary of a PATROL event

Format

`add_diary(`*`eventID`*`,`*`text`*`,[`*`user_name`*`])`

Parameter

Parameter	Definition
`eventID`	unique number returned by event_trigger() on page 1-146 or event_trigger2() on page 1-149
text	text that you want to add to the diary of a PATROL event
`user_name`	optional parameter that you can use to add a user ID to the event diary

Description

The add_diary() function enters text in the diary of a PATROL event. It writes information to the event diary in the following format:

`user_name date_and_time text`

After using this function, you can verify the event diary information by viewing the event in the PATROL Event Manager.

Note

This function is similar to the PemnAddDiary() function available in the PATROL API.

Example

This script demonstrates the add_diary() function.

```
save_event_ID = event_trigger("STD", "41", "INFORMATION", "3", "testing add
diary");

add_diary(save_event_ID,"This comment should appear in the event diary.",
"jsmith");
```

Running the preceding script should put the following information in the event diary:

```
jsmith Mon Jan 08 15:02:38 This comment should appear in the event diary.
```

BMC Software, Inc., Confidential and Proprietary Information

annotate()

Provide contextual information for a PATROL parameter

Format

`annotate(`*`parameter`*`,`*`format`*`,`*`data1`*`,[...,`*`datan`*`])`

Parameters

Parameter	Definition
`parameter`	name of the parameter for which annotation information is specified The parameter name can be an absolute path name, or a path name relative to the current or other parameter name. If you are using the %Text or %InfoBox formats you can specify the NULL string `""` or `"."`, and the annotate() function uses the current parameter name or the parameter name for which the current recovery script is running. However, if you are using the %Response format, you must specify *parameter*.
`format`	mnemonic indicating the type of display for the specified annotation information See “Specifying the annotate() Function Output Format” on page 1-16 for more information. Valid Values • %Text—display annotation data as a character string in a text editor window • %InfoBox—display annotation data in the PATROL InfoBox style • %Response—display text as a read-only response() function dialog box **Default** %Text
`datan`	One or more optional text strings that specify the annotation information. The *datan* parameters are interpreted differently depending on *format*.

BMC Software, Inc., Confidential and Proprietary Information

Description

The annotate() function instructs the PATROL Agent to save information represented by *datan* for *parameter* in the history database for later viewing as *format*.

The annotate() function allows a Knowledge Module to annotate a parameter value with contextual information that allows a user to easily identify the cause of an unusual condition. The additional information, represented by the *datan* parameter, is stored historically in the PATROL Agent history database and is subsequently viewable from the PATROL Console, from a PSL script, or through archiving.

The PATROL Console can view annotation information for the current parameter value or historical values. The PATROL Console accesses the annotation information using drill-down on data points or options (depending on the context). The annotation information is presented either as plain text, a PATROL InfoBox, or as a PSL response() function dialog.

Annotation and History Retention

The PATROL Event Manager/Agent retains annotations for the same duration as the parameter values that they accompany. Annotations are kept in the history database using the usual history retention settings in the Knowledge Module and PATROL Agent configurations. Annotations are purged when the accompanying parameter value is purged. Thus, when specified, every data point retained in the history database has an accompanying annotation.

The annotate() function succeeds even when the history retention is zero because it stores the current annotation in memory with the current parameter value. There is no need to pretext the value of the history_get_retention() function before calling the annotate() function. In this case, however, multiple annotations are not stored, nor are annotations persistent.

BMC Software, Inc., Confidential and Proprietary Information

annotate() Return Codes

If successful, the annotate() command returns the NULL string and sets the PSL variable errno = 0 (E_PSL_NO_ERROR). Upon failure, the annotate() function returns the NULL string, can return the message FAILED to the computer icon display window, and sets the PSL variable errno.

The annotate() function return value lends itself to the usage:

```
ann_return = annotate(parameter,format);
if (ann_return) {
   print("Error from annotate() : ",ann_return,”\n”);
}
```

The following are specific annotate() function error conditions.

Cannot find runtime object

The annotate() function returns the message Cannot find runtime object and sets errno = 103 (E_PSL_NO_RTCELL) when *parameter* does not exist or is not a parameter object.

annotate is called before any data point generated

The annotate() function returns the message annotate called before any data point generated and sets errno = 115 (E_PSL_ANN_TOO_EARLY) when it attempts to annotate a *parameter* that has not yet stored a value. This can occur when the annotate() function is called before a set() function can set the *parameter* value.

Cannot save annotation information

The annotate() function returns the message Cannot save annotation information and sets errno = 116 (E_PSL_ANN_SAVE_FAILED) when an attempt to save the annotation data fails. This can be caused by permission restrictions on the history database. This error can also be the result of an internal PATROL error.

BMC Software, Inc., Confidential and Proprietary Information

Specifying the annotate() Function Output Format

The annotate() function *format* parameter specifies the style of the annotation output on a PATROL Console and determines how the annotate() function interprets the *datan* strings.

"%Text"—Plain Text Output

When *format* = "%Text" or *format* = ""is specified, the annotate() function treats the *datan* annotations as a single string of text in a text editor window. Each *datan* can be a multiline string containing newline (\n) characters. The *format* = "%Text" option is useful for displaying long results such as command output or error log files.

Note

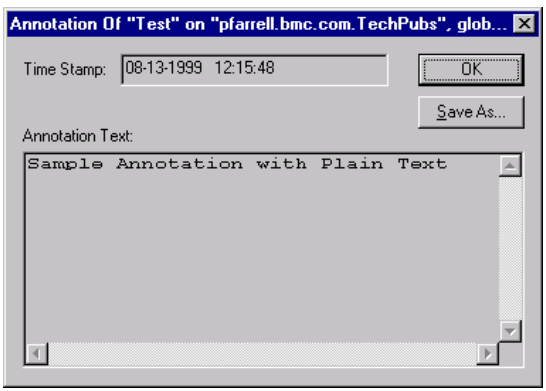
Long annotations use more disk space in the PATROL Agent history annotation database.

The following example creates a plain-text annotation:

```
annotate("","","Sample Annotation with Plain Text");
```

When this annotation is viewed on a PATROL Console, a plain-text dialog box displays as shown in Figure 1-1:

Figure 1-1 Plain-Text Annotation Dialog Box



"%InfoBox"—PATROL Console InfoBox Output

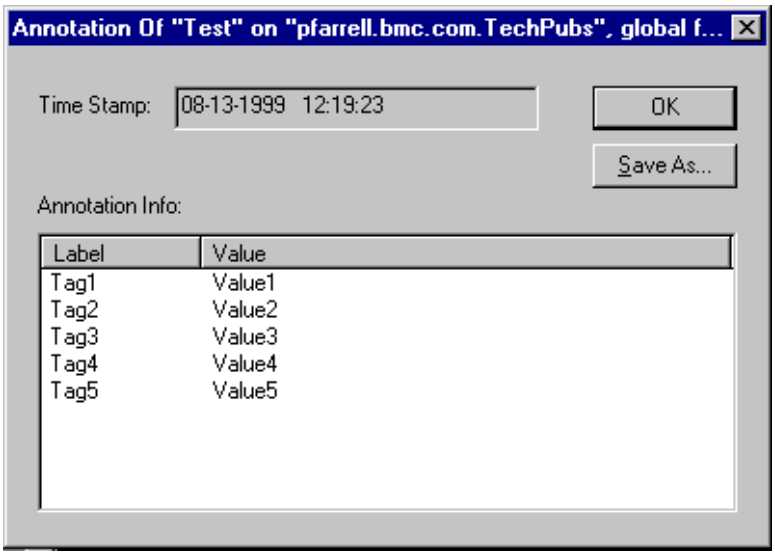
When format = "%InfoBox" is specified, the annotate() function treats the *datan* annotations as PATROL Console InfoBox item strings in the form "*tagn*\n*valuen*"where *tagn* is the InfoBox item label and *valuen* is the InfoBox item label.

The following example creates an InfoBox annotation:

```
annotate("","%InfoBox","Tag1\nValue1",
"Tag2\nValue2",
"Tag3\nValue3",
"Tag4\nValue4",
"Tag5\nValue5"
);
```

When this annotation is viewed on a PATROL Console, an InfoBox dialog box displays as shown in Figure 1-2:

Figure 1-2 InfoBox Annotation Dialog Box



"%Response"—PSL response() Function Dialog Output

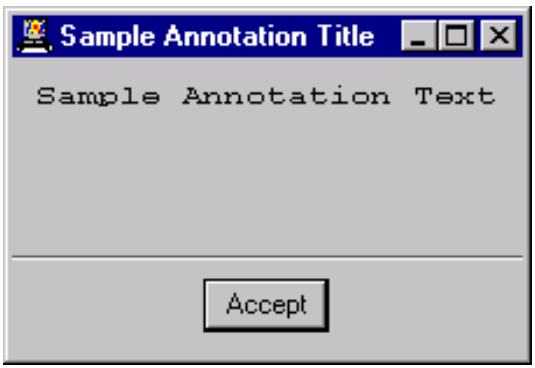
When format = "%InfoBox" is specified, the annotate() function treats the *datan* annotations as PSL response() function element definitions. (See “response()” on page 1-326 for more information about the response() function element definitions.) The response() function dialog generated by the annotate() function is read-only.

The following example creates a response() function annotation:

```
annotate("Parameter","%Response",
"Sample Annotation Title","-1",
["h=100",
"w=100",
"b=Patrol",
"e=1",
"o=Accept",
"c=Cancel",
"N=0",
"A=0",
"B=0"],
[R_LABEL_CENTER, "Sample Annotation Text"]
);
```

When this annotation is viewed on a PATROL Console, a response() function dialog box displays as shown in Figure 1-3:

Figure 1-3 response() Function Annotation Dialog Box



annotate_get()

Return the annotation text for a parameter value

Format

`annotate_get(`*`parameter`*`,[`*`timestamp`*`])`

Parameters

Parameter	Definition
parameter	name of the parameter for which annotation information is specified The parameter name can be an absolute path name, or a path name relative to the current or other parameter name. If you specify the NULL string `""` or the string `"."`, the `annotate()` function uses the current parameter name or the parameter name for which the current recovery script is running.
timestamp	a string that identifies the timestamp of the desired annotation The timestamp is the timestamp of the set() function that set the parameter value and not the timestamp of the annotate() function that created the annotation. The timestamp is the number of seconds that have elapsed since 00:00:00 GMT, Jan 01, 1970. The timestamp of the data points for a parameter can be obtained with the PSL history() function. **Default** If the *timestamp* parameter is omitted, the most recent data point for *parameter*. (This may not be the same as the most recent call to the `annotate()` function as there may have been another data point stored by a `set()` function.)

BMC Software, Inc., Confidential and Proprietary Information

Description

The annotate_get() function returns the annotation for the *parameter* value at *timestamp*. The annotate_get() function is designed for the more advanced uses of the annotate() function, such as those that require verification of whether an annotation already exists.

Return Values

If successful, the annotate_get() function sets the PSL variable errno = 0 (E_PSL_NO_ERROR) and returns the annotation string that consists of the data arguments specified in the annotate() function that created the annotation. There are three types of strings that the annotate_get() function will return.

Plain-Text Annotation Text

If the `annotate()` function created a plain-text annotation the annotation is returned in a form similar to the following text:

```
\Sample Annotationwith Plain Text\
```

InfoBox Annotation Text

If the `annotate()` function created a plain-text annotation the annotation is returned in a form similar to the following text:

```
%InfoBox\Tag1
Value1\Tag2
Value2\Tag3
Value3\Tag4
Value4\Tag5
Value5\
```

response() Function Text

If the `annotate()` function created a response() function annotation, the annotation is returned in a form similar to the following text:

```
%Response\Sample Annotation\-1\h=100
w=100
b=Patrol
e=1
o=Accept
c=Cancel
N=0
A=0
B=0\16
2\1
Sample Annotation Text\
```

Errors

The following are specific annotate_get() function error conditions.

Invalid or Non-parameter Object

The annotate_get() function returns the NULL string and sets errno = 103 (E_PSL_NO_RTCELL) when the parameter does not exist or is not a parameter object.

No Annotation for Parameter

The annotate_get() function returns the NULL string and sets errno = 105 (E_PSL_ANN_NO_HISTORY_RETENTION) when *parameter* is valid but

- *timestamp* is omitted and no annotations exist for any value of *parameter* in the PATROL Agent history database
- *timestamp* is specified and no annotation exists for the value of *parameter* at *timestamp*

Note

If you specify the NULL string for the *timestamp* parameter, the annotate_get() function sets the errno value to 105 because no timestamp matches the NULL string. Omit the *timestamp* parameter if you want to obtain the annotation of the last data point for a parameter.

BMC Software, Inc., Confidential and Proprietary Information

asctime()

Return the date and time as a character string

Format

`asctime(`*`clock`*`,[`*`format`*`])`

Parameters

Parameter	Definition
clock	a reference to the clock or timer whose value should be converted to a character string The *clock* is most commonly `time()`.
format	Optional format symbol for the `asctime()` output string. See Table 1-2 for some common field symbols. **Default** 24-character string with the following format: Sun Sep 16 01:03:52 1973

Description

The `asctime()` function returns the date/time of *clock* as a character string. It is equivalent to the C library `asctime()` function except that the PSL `asctime()` function adjusts the value to the local timezone. If *format* is given, `asctime()` returns the date/time string in the specified format.

Note

The *format* field specifiers are similar to those used in the C library `strftime()` function and are dependent on the operating system. Refer to the C programming reference for your operating system for *format* field specifiers.

BMC Software, Inc., Confidential and Proprietary Information

Table 1-2 Format Symbols for asctime() (Part 1 of 3)

Format	Description
%%	this symbol allows you to use a percent sign (%) in the format of a date string
%a	locale's abbreviated name of the day of week
%A	locale's full name of the day of week
%b	locale's abbreviated name of the month
%B	locale's full name of the month
%c	locale's appropriate date and time representation
%C	data and time as %c
%d	day of month [1,31]; single digits are preceded by 0
%D	date as %m/%d/%y
%e	day of month [1,31]; single digits are preceded by a space
%h	locale's abbreviated name of the month
%H	hour (24-hour clock) [0,23]; single digits are preceded by 0
%I	hour (12-hour clock) [1,12]; single digits are preceded by 0
%j	day of year [1,366]; single digits are preceded by 0
%k	hour (24-hour clock) [0,23]; single digits are preceded by a space
%l	hour (12-hour clock) [1,12]; single digits are preceded by a space
%m	month as a decimal number [1,12]; single digits are preceded by 0
%M	minute [0,59]; leading zero is permitted but not required
%n	insert a new line
%p	locale's equivalent of either a.m. or p.m.
%r	appropriate time representation in 12-hour clock format with %p
%R	time as %H:%M
%S	seconds [0,61]
%t	insert a tab
%T	time as %H:%M:%S
%u	day of week as a decimal number [1,7], with 1 representing Monday
%U	week of the year as a decimal number [0,53], with Sunday as the first day of week 1

BMC Software, Inc., Confidential and Proprietary Information

Table 1-2 Format Symbols for asctime() (Part 2 of 3)

Format	Description
%V	week of the year as a decimal number [01,53], with Monday as the first day of the week If the week containing 1 January has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1.
%w	day of week as a decimal number [0,6], with 0 representing Sunday
%W	week of the year as a decimal number [0,53], with Monday as the first day of week 1
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year within century [0,99]
%Y	year, including the century (for example 1993)
%Z	abbreviated or full name of time zone, or no bytes if no information of the time zone exists
%Ec	locale's alternative appropriate date and time representation
%EC	name of the base year (period) in the locale's alternative representation
%Ex	locale's alternative date representation
%EX	locale's alternative time representation
%Ey	offset from %EC (year only) in the locale's alternative representation
%EY	alternative representation of the year in full
%Od	day of the month using the locale's alternative numeric symbols
%Oe	same as %Od
%OH	hour (24-hour clock) using the locale's alternative numeric symbols
%OI	hour (12-hour clock) using the locale's alternative numeric symbols
%Om	month using the locale's alternative numeric symbols
%OM	minutes using the locale's alternative numeric symbols
%OS	seconds using the locale's alternative numeric symbols
%OU	week of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols
%Ow	day of week (Sunday=0) using the locale's alternative numeric symbols

BMC Software, Inc., Confidential and Proprietary Information

Table 1-2 Format Symbols for asctime() (Part 3 of 3)

Format	Description
%OW	week of the year (Monday as the first day of the week) using the locale's alternative numeric symbols
%Oy	year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols

Example

The following PSL script shows how the asctime() function can translate the computer system time value into a familiar Gregorian date and time:

```
raw_time = time();

default_output = asctime(raw_time);
printf("raw time is %ld, default asctime() output is
%s\n",raw_time,default_output);

output_with_format = asctime(raw_time,"Day is %A, Month is %B, Time is
%X\nTimezone is %Z");

printf("%s\n",output_with_format);
This example returns the following output:
raw time is 825948788, default asctime() output is Mon Mar  4 08:13:08 1996
Day is Monday, Month is March, Time is 08:13:08
Timezone is CST
```

BMC Software, Inc., Confidential and Proprietary Information

asin()

Return the arcsine of the argument

Format

`asin(`*`sine`*`)`

Parameter

Parameter	Definition
sine	sine argument **Valid Values** $-1 \le sine \le 1$

Description

The `asin()` function returns the arcsine of *sine*, that is, the length in radians of the arc whose sine is *sine*. The output range for the `asin()` function is $-\pi/2 \le \text{asin}() \le \pi/2$.

Example

The following PSL script uses the asin() function to confirm the trigonometric identity Arcsin x + Arccos $x = \pi/2$:

```
PI = 3.141593;
HALFPI = PI / 2;
print("Arcsine + Arccosine Identity Verification\n\n");
print("  Sine     Expected   Returned\n");
print("---------  ---------  ---------\n");
sine = -0.9;
while (sine < 1.0) {
   printf("%+8.6f  %+8.6f  %+8.6f\n",sine,HALFPI,asin(sine) + acos(sine));
   sine = sine + 0.1; }
```

BMC Software, Inc., Confidential and Proprietary Information

atan()

Return the arctangent of the argument

Format

```
atan(tangent)
```

Parameter

Parameter	Definition
tangent	tangent argument **Valid Values** $-\infty \le tangent \le \infty$

Description

The `atan()` function returns the arctangent of *tangent*, that is, the length in radians of the arc whose tangent is *tangent*. The output range for the `atan()` function is $-\pi/2 \le \text{atan}() \le \pi/2$.

Example

The following example uses atan() to verify the trigonometric identity:

$$\sin(\text{atan}(x)) = \frac{x}{\sqrt{1 + x^2}}$$

```
print("Arcsine(Arctangent) Identity Verification\n\n");
print("  tan()     Expected   Returned\n");
print("---------  ---------  ---------\n");
tangent = -1.5;
while (tangent <= 1.5) {
   expect = tangent / sqrt(1 + pow(tangent,2));
   printf("%+8.6f  %+8.6f  %+8.6f\n",tangent,expect,sin(atan(tangent)));
   tangent = tangent + 0.1; }
```

BMC Software, Inc., Confidential and Proprietary Information

atexit()

Specify a user-defined PSL function to be called upon the normal termination of a PSL process

Format

`atexit(`*`function_name`*`)`

Parameter

Parameter	Definition
function_name	a string specifying the name of a user-defined PSL function you want to call

Description

Use the `atexit()` function to specify a user-defined PSL function to be called upon the normal termination of a PSL process. You must define the specified function *before* the `atexit()` call, and it must take no arguments.

The `atexit()` function is similar to the C `atexit()` function. The major difference is that, with the PSL `atexit()` function, only one function can be executed upon normal termination of the process.

The `atexit()` function returns one of the following values, indicating whether the call succeeded:

0 Success
1 User function not defined
2 User function accepts too many arguments
3 Out of memory

If the return value is non-zero, then the call to `atexit()` failed, and the function will *not* be called upon normal termination.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the `atexit()` function:

```
function my_exit_handler() {
   # This function must take no arguments and be
   # defined before the atexit() call.

   print("Hello from 'my_exit_handler'.\n");
}

function main() {
   print("Before the call to atexit()\n");
   atexit("my_exit_handler");
   print("After the call to atexit()\n");
                }
```

This example returns the following output:

```
Before the call to atexit()
After the call to atexit()
Hello from 'my_exit_handler'.
```

Note

The PSL compiler might warn that the function my_exit_handler was not called or exported. This warning can safely be ignored.

BMC Software, Inc., Confidential and Proprietary Information

batch_set

Assigns values to a series of variables

Format

```
batch_set(data,[set_delimiter],[element_delimiter],
   [time_stamp])
```

Parameter

Parameter	Definition
data	list of variables and the values to be assigned to these variables
set_delimiter	character used in *data* to separate one set from another In this function, a set is a variable and the value to be assigned to the variable. If the \n character is the *set_delimiter*, *data* may look like `"/a,5\n/b,7"`. The default value of *set_delimiter* is \n.
`element_delimiter`	character used in *data* to separate a variable from the value to be assigned to the variable A comma (,) is the default value. See the example for *set_delimiter* in the above cell, which shows default input values.
time_stamp	time that the variable was set PATROL automatically records the time that a variable is set, and it may display the time as a data point in a graph or chart generated by the PATROL Console. The *time_stamp* parameter gives you the option of overriding the value that PATROL would assign. This parameter is entered as the number of seconds that have elapsed since 00:00:00 GMT, Jan 01, 1970.

BMC Software, Inc., Confidential and Proprietary Information

Description

This function is similar to the set() function except that batch_set() allows you to assign values to many variables at one time.

The following table explains the return values for this function.

Table 1-3 Return Values for batch_set()

Return Value	Description
`0`	function completed successfully
`1`	function terminated because *set_delimiter* and *element_delimiter* have the same value
`2\nvariable1,errno1` `[... \nvariablen,`*`errnon`*`]`	no value was assigned to at least one variable If batch_set() cannot set a variable, it returns error information for that variable and continues setting the remaining variables. The function returns a two (2) to indicate that one or more variables could not be set. For each variable that could not be set, the function also returns the name of the variable and the PSL errno value. For example, a return value of `"2\n/ipAddress,60"` would mean that the variable */ipAddress* could not be set because it is not writable (errno 60).

Example

The following example shows that you can write code that evaluates the return value as a number. Even though a failure of the batch_set() function returns a list, the statement works because it simply tests whether the value is a non-zero number.

```
if (batch_set(data) != 0) {
# handle failure
}
```

The following shows two more examples.

BMC Software, Inc., Confidential and Proprietary Information

```
data = "/a,1\n/b,2,".time()."\n/c,3\n/ipAddress,hello
   world";
result = batch_set(data);
save_errno = errno;
printf("errno=%s\n", save_errno);
printf(">%s<\n", result);
printf("a=%s\n", get("/a"));
printf("b=%s\n", get("/b"));
printf("c=%s\n", get("/c"));
```

```
data = "/a|10@/b|20|".time()."@/c|30@/ipAddress|hello
   world";
result = batch_set(data, "@", "|");
save_errno = errno;
printf("errno=%s\n", save_errno);
printf(">%s<\n", result);
printf("a=%s\n", get("/a"));
printf("b=%s\n", get("/b"));
printf("c=%s\n", get("/c"));
```

BMC Software, Inc., Confidential and Proprietary Information

blackout()

Hide PATROL object state changes for a specific period of time

Syntax

`blackout([object],[period])`

Parameters

Parameter	Definition
object	name of the PATROL application or instance that has its state changes hidden If *object* is omitted, *period* must also be omitted.
period	length of time the PATROL application or instance remains blacked out. If *period* is specified, *object* must also be specified. **Valid Values** A character string in the following format: dd:hh:mm where *dd* is days, *hh* is hours (0 – 23), and *mm* is minutes (0 – 59)

Description

The blackout( ) function starts and stops the blacking out of state changes (that is, OK ⇔ WARN ⇔ ALARM transitions) for an instance or application displayed on the PATROL Console.

BMC Software, Inc., Confidential and Proprietary Information

The effect of the blackout( ) function is similar to that of the **Alarm Snooze** option on the PATROL Console object menus. However, while the **Alarm Snooze** option clears alarm states for the specified period on a specific console, the blackout() function preserves the current state of the object for the specified period on all consoles because it is applied at the agent.

When the blackout() function is specified for an object that is currently not available, but will be online in the future, the blackout will take effect immediately upon the creation of the object.

The blackout() function performs the following actions depending on how you specify its parameters:

- The blackout() function with no parameters returns a list of all PATROL objects that are currently blacked out. The object names in the returned list are separated by new-line characters (\n).

- The blackout (*object*) function where *object* is an instance returns the blackout status for single instance as the following character string:

  instance\n
  blackoutend

  Where *instance* is the instance name and *blackoutend* is the system time when the blackout is to end.

- The blackout (*object*) function where *object* is an application returns the blackout status for all instances that belong to the application as the following character string:

  instance\n
  blackoutend\n
  instance\n
  blackoutend\n
  ...
  ...
  instance\n
  blackoutend

BMC Software, Inc., Confidential and Proprietary Information

Where *instance* is the instance name and *blackoutend* is the system time when the blackout is to end.

- The blackout (*object*, *period*) function sets the blackout for *object* to *period* and returns the string `OK` if successful.

  When the requested *object* is an application, the blackout() function will blackout all instances that belong to the application and reset any existing blackout periods for this application to the requested *period*.

  If an application is blacked out, the blackout() function can be applied to an instance belonging to the application, but the blackout period may only be increased. The blackout period for an instance cannot be shorter than the blackout period of its application.

- The blackout (*object*,"") function ends the blackout of *object* and returns the string OK if successful.

  The blackout of an instance cannot be ended if its application has a blackout in effect. If an instance has a longer blackout period than its application, and the blackout is ended for the instance, the blackout period will be reset to the applications blackout period.

BMC Software, Inc., Confidential and Proprietary Information

Examples

The following examples demonstrate how the blackout() function performs differently when the *object* is an instance and when it is an application.

Instance–The following PSL statements use the blackout() function to perform the following operations:

- set a blackout for an instance
- list all instances that are blacked out
- return the blackout information for an instance
- end the blackout for instance

```
function main() {
   instance = "/FILESYSTEM/tmp_mnt-home-dherr";
   print("Test instance is ",instance,"\n");

   print("Blacking out instance . . . ",blackout(instance,"00:00:05"),"\n");

   print("Return all blacked out objects . . .\n",blackout(),"\n");

   print("Return blackout information for ",instance,"\n");
   output = blackout(instance);
   instance_name = nthlinef(output,1);
   blackout_time = nthlinef(output,2);
   print("        Instance : ",instance_name);
   print("Blackout Expires : ",asctime(blackout_time),"\n");

   print("End blackout for ",instance," . . . ",blackout(instance,""),"\n");
}
```

The above PSL statements return the following output:

```
Test instance is /FILESYSTEM/tmp_mnt-home-dherr
Blacking out instance . . . OK
Return all blacked out objects . . .
/FILESYSTEM/tmp_mnt-home-dherr

Return blackout information for /FILESYSTEM/tmp_mnt-home-dherr
        Instance : /FILESYSTEM/tmp_mnt-home-dherr
Blackout Expires : Mon May 05 15:12:28 1998
End blackout for /FILESYSTEM/tmp_mnt-home-dherr . . . OK
```

BMC Software, Inc., Confidential and Proprietary Information

Application–the following PSL statements use the blackout() function to perform the following operations:

- set a blackout for an application
- list all applications that are blacked out
- return blackout information for an application
- end the blackout for application

```
application = "/FILESYSTEM/";
print("Test application is ",application,"\n");
print("Blacking out...",blackout(application,"00:00:05"),"\n");
print("Return all blacked out objects . . .\n",blackout(),"\n");
print("Return blackout information for ",application,"\n");
output = blackout(application);
for(i =1; i < lines(output); i+=2) {
   instance_name = nthlinef(output,i);
   blackout_time = nthlinef(output,i+1);
   print("        Instance : ",instance_name);
   print("Blackout Expires : ",asctime(blackout_time),"\n");
   }
print("End blackout for ",application,"...",blackout(application,""),"\n");
```

The above PSL statements return the following output:

```
Test application is /FILESYSTEM/
Blacking out . . . OK
Return all blacked out objects . . .
/FILESYSTEM/stand
/FILESYSTEM/usr
/FILESYSTEM/tmp
/FILESYSTEM/local

Return blackout information for /FILESYSTEM/
       Instance : /FILESYSTEM/stand
Blackout Expires : Mon May 05 15:12:28 1998
       Instance : /FILESYSTEM/usr
Blackout Expires : Mon May 05 15:12:28 1998
       Instance : /FILESYSTEM/tmp
Blackout Expires : Mon May 05 15:12:28 1998
       Instance : /FILESYSTEM/local
Blackout Expires : Mon May 05 15:12:28 1998

End blackout for /FILESYSTEM/. . . OK
```

BMC Software, Inc., Confidential and Proprietary Information

cat()

Return the content of a file as a single text string

Format

`cat(`*`filename`*`)`

Parameter

Parameter	Definition
filename	name of the file whose contents are to be returned

Description

The `cat()` function returns the contents of file *filename* as a single string or the NULL string on error. Newlines are preserved so that the *foreach* statement can be used to process the returned string as a list of the lines in *filename*.

The `cat()` function can return the following PSL errno values:

errno Value	Description
1	E_PSL_CAT_MEMORY_FAILURE memory allocation failure
2	E_PSL_CAT_READ_ERROR read error
29	E_PSL_FILE_CANNOT_OPEN could not open file
30	E_PSL_FILE_NOT_FOUND file not found
72	E_PSL_FOPEN_BAD_ACCOUNT insufficient file permissions:

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL statements list the names of users listed in the Unix system password file:

```
people = cat("/etc/passwd");
foreach person (people) {
   name = ntharg(person, 1, ":");
   print("name of person is:", name, "\n");
}
```

ceil()

Return the smallest integer that is not less than the argument

Format

`ceil(`*`arg`*`)`

Parameter

Parameter	Definition
arg	numeric argument whose least integer upper bound is to be determined

Description

The `ceil()` function returns the smallest integer that is not less than *arg*, that is, the least integer upper bound for *arg*.

The `ceil()` function and the `floor()` function together bracket *arg* such that the following are true:

If *arg* is an integer,
ceil(*arg*) = *arg* = floor(*arg*)

If *arg* is not an integer:
floor(*arg*) < *arg* < ceil(*arg*) and ceil(*arg*) = floor(*arg*) + 1

Example

The following example shows how to use the `ceil()` function:

```
any_float = 8.4;
ceil_any_float = ceil(any_float);
printf("Ceiling of %f is %d\n",anyfloat,ceil_any_float);
```

BMC Software, Inc., Confidential and Proprietary Information

chan_exists()

Verify that a process or file channel exists

Format

chan_exists(*channel*)

Parameter

Parameter	Definition
channel	the process or file I/O channel name (shared channels) or number (local channels) that is being verified

Description

The `chan_exists()` function returns 1 if the local or shared *channel* exists and 0 if it does not.

The `chan_exists()` function return value can be used with condition variables for synchronizing one PSL process to wait until another has opened a channel using either the `popen()` or `fopen()` function.

Example

The following PSL statements use the `chan_exists()` function to verify that a file channel successfully opened:

```
chan = fopen("/etc/passwd","r");
if (chan_exists(chan)) {
   print("Channel successfully opened\n");
} else {
   print("Error opening file\n");
}
close(chan);
```

BMC Software, Inc., Confidential and Proprietary Information

change_state()

Change the rule state of a PATROL application instance

Format

`change_state(`*`object`*`,`*`state`*`,[`*`description`*`])`

Parameters

Parameter	Definition
object	name of the application instance whose state is to be changed
state	operational state to which *object* will be set **Valid Values** • OFFLINE • OK • WARN • ALARM **Note:** Typically you should use the OK state. Using any other state causes PATROL to suspend the parameters that the object may have.
description	optional text string that can be used to explain the state change action The text string must be enclosed in double quotation marks. The text string is recorded in the Event Diary of the State Change Event in the Event Manager. **Default** NULL string

BMC Software, Inc., Confidential and Proprietary Information

Description

The `change_state()` function changes the rule state of application instance *object* to *state*. The `change_state()` function returns the new state or the NULL string on error.

Rule State and Combined State

Each PATROL object has two state values associated with it: a *rule* state and a *combined* state. The rule state is the state of the object. The combined state is the state of the object and all its children.

When an object changes state and propagates that state change to its parent, it can change the combined state of its parent but not the rule state. However, if you change the state of an object using the `change_state()` function, you change the rule state of that object.

Setting the rule state of an object via the `change_state()` function *is not* equivalent to setting the combined state via child ⇒ parent state propagation. If you change the rule state of an application instance to ALARM (or OFFLINE) via the `change_state()` function, the PATROL Agent assumes that the application instance is in critical condition and deactivates all its parameters while it is in that state. But if you allow an ALARM state to propagate from parameter to application instance, the application instance will show ALARM as its combined state, but if its rule state is not in ALARM the PATROL Agent will not deactivate its parameters.

Tip

You should never use the `change_state()` function to set the state of the parent object.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL statement changes the state of object `person` to the ALARM state and issues a message to the Event Manager Event Diary:

```
# change state of instance <person>
change_state(person,ALARM,"User close to process quota");
```

BMC Software, Inc., Confidential and Proprietary Information

chart()

Pop up, destroy, or print a chart (graph) on the PATROL Console

Format

`chart(`*`command`*`,`*`gdf_string`*`,[`*`ID`*`])`

Parameters

Parameter	Definition
`command`	built-in constants (symbolic names) for the different chart commands **Valid Values** CHART_LOAD–loads a specified chart CHART_PRINT–prints a loaded chart CHART_DESTROY–destroys a loaded chart
`gdf_string`	a string describing the chart For more information, see the description of the GDF (Graph Definition Format) syntax on page 1-47.
`ID`	name identifying the chart on the console The ID is returned by the CHART_LOAD command, and it is required for CHART_PRINT and CHART_DESTROY commands.

BMC Software, Inc., Confidential and Proprietary Information

Description

The `chart()` function loads (displays), prints, and destroys charts (also called graphs) on the PATROL Console.

Loading a Chart

Using CHART_LOAD for the *command* parameter loads a chart on the PATROL Console. The contents of the chart are specified in a Graph Description Format (GDF) string. See the description of the GDF syntax on page 1-47. The following example shows the use of the CHART_LOAD command:

```
chart(CHART_LOAD,gdf_string);
```

The GDF information can be specified in the function call, supplied as a variable, or specified in an external file. See “Specifying GDF data in chart() function calls” on page 1-52 for more information on specifying GDF information in chart() function calls.

The return value of the chart() function call with the CHART_LOAD command is the handle for the chart, and it is required as the *ID* parameter for the CHART_PRINT and CHART_DESTROY operations.

Printing a Loaded Chart

Using CHART_PRINT for the *command* parameter prints a chart that is loaded on the PATROL Console. When the CHART_PRINT command is used, the PATROL Console displays a print dialog box for the user to print the chart. The following example shows the use of the CHART_PRINT command:

```
chart(CHART_PRINT,"",ID);
```

BMC Software, Inc., Confidential and Proprietary Information

Destroying a Loaded Chart

Using CHART_DESTROY for the *command* parameter destroys a chart that is loaded on the PATROL Console. When the chart is destroyed, it disappears from the console display, and it is removed from the console memory. The following example shows the use of the CHART_PRINT command:

```
chart(CHART_DESTROY,"",ID);
```

Errors

The `chart()` function returns the following PSL errno values on failure:

errno Value	Description
93	E_PSL_NO_SUCH_ID The required ID was not supplied or it is invalid.
120	E_PSL_BAD_CHART_COMMAND An invalid command was supplied.

Graph Definition Format (GDF) Syntax

The GDF definition for a PATROL graph consists of comments and keywords that define the graph and the data values for the graph.

GDF Comments

Comments in a GDF definition are usually used when a GDF definition is stored in an external file. A GDF comment must have an exclamation mark (!) as its first character, and a comment can contain any character except the NULL character or a new-line (\n) character.

BMC Software, Inc., Confidential and Proprietary Information

GDF Chart Definition Keywords

GDF provides a set of keywords that are used to define the graph. The keywords values are specified using the following format:

keyword:*value*

The GDF keyword values can contain any combination of letters, numbers, spaces, underscores (_), and dashes(-). The keywords can be separated into two types graph definition and data definition keywords.

Each keyword specification is either terminated with a new-line (\n) character or a line break in an external file.

Graph Definition Keywords

There are several keywords that define the appearance of the chart on the PATROL Console. These keywords control the titles, scales, colors of the chart. The following table describes the GDF graph definition keywords:

Keyword Format	Description
WindowTitle:*Window_Title*	the title of the window or icon
ChartTitle:*Chart_Title*	the title of the chart
XAxisLabel:*X_Axis_Label*	the label for the x-axis of the chart
YAxisLabel:*Y_Axis_Label*	the label for the y-axis of the chart
ChartTemplate:*File_Name*	a file that contains the chart template The *File_Name* may contain the path to the chart template file. The chart template file can be created by using the PATROL Developer Console and saving your chart customizations. See your PATROL Console documentation for more information on customizing PATROL charts.

Data Definition Keywords

GDF provides two keywords for defining the data that is displayed in the chart:

Keyword Format	Description
`StaticData2D`	defines static data Static data is data that is defined before the chart is displayed and is not updated as the chart is displayed.
`PatrolConsole`	defines dynamic data Dynamic data is data that is read directly from a PATROL parameter, and it is updated on the chart as the parameter is updated.

BMC Software, Inc., Confidential and Proprietary Information

Static Data

Static data is created by supplying the data name and the data parameters to the chart() function in the GDF string. The data portion of the GDF string for a static chart contains the data name and one or more data points in the following format:

`StaticData2D:`*`name, time value,...,timen valuen`*`\n`

Variable	Description
name	the name of the value that is graphed The *name* appears in the name column of the graph legend.
time value,... *timen valuen*	one or more pairs of time and value that are graphed The time is the current local system time specified as the number of seconds that have elapsed since 00:00:00 GMT, Jan 01, 1970. The *time value* pairs are separated by commas, and the *time* and *value* in the pair are separated by spaces.

The following example is the data portion of a GDF string for a static chart:

```
data="StaticData2D:Value Name, 956159608 10.5, 956159668 20.4\n";
```

Dynamic Data

Dynamic data is created by supplying a host name and a parameter name to the chart() function in the GDF string. The data is then read from the host machine and displayed on the graph. A dynamic data chart is updated with new data points by the agent as long as the chart is active.

The data portion of the GDF string for a dynamic chart contains the host name and the parameter name in the following format:

`PatrolConsole:`*`host_name, parameter_name`*`\n`

Variable	Description
host_name	the name of the host that the data is displayed for in the chart The *host_name* must be specified *in the same format* as it appears in the tree view of the PATROL Console displaying the chart. There are three ways that a user can enter the host in the tree view of the PATROL Console: • simple name–*hostname* • qualified name–*hostname.domain.com* • numerical address–*nnn.nnn.nnn.nnn* The *host_name* specified in the chart() function call is not case sensitive. **Note** There are a couple of points to consider about the host name in the tree view of the PATROL Console: • For the chart() function to find the host and properly display the data, the host name must be in all upper-case or all lower-case characters on the PATROL Console. • There is no way to determine the host name entry in the tree view of the PATROL Console using PSL. The get("hostname") function call always returns the simple host name in all upper-case or all lower-case characters.
parameter_name	the name of the parameter that has its values displayed in the chart The parameter name must include the entire PATROL path name that follows the host name.

The following example is the data portion of a GDF string for a dynamic chart:

```
data="PatrolConsole:pfarrell,/NT_MEMORY/NT_MEMORY/MEMmemPagesInputPerSec\n"
```

Specifying GDF data in chart() function calls

The GDF definition for the chart is usually specified using on of the following techniques:

- the GDF string is specified in the chart() function call

  id = chart(CHART_LOAD,”StaticData2D:CPU, 956159600 10.5\n”);

- the GDF string is assigned to a variable that is used in the chart() function call

  gdf_string=”StaticData2D:CPU, 956159600 10.5, 956160200 20.4\n”;
  id = chart(CHART_LOAD,gdf_string);

- the GDF string is specified in an external file

  id = chart(CHART_LOAD,cat("file.gdf"));

  The following example shows the syntax of an external GDF file:

```
WindowTitle: Data Analysis from Patrol 3.4

!Here is a comment
!Use current directory file named "example.ctf" as template file
ChartTemplate: example.ctf
ChartTitle: hostname CPU and Memory Usage
XAxisLabel: April 19, 2000
YAxisLabel: %

!Then add chart data
StaticData2D:CPU for hostname, 956159600 10.5, 956160200 20.4, 956160800 40.4
StaticData2D:Memory for hostname, 956159600 50.5, 956160200 60.7, 956160800 71.3
```

Examples

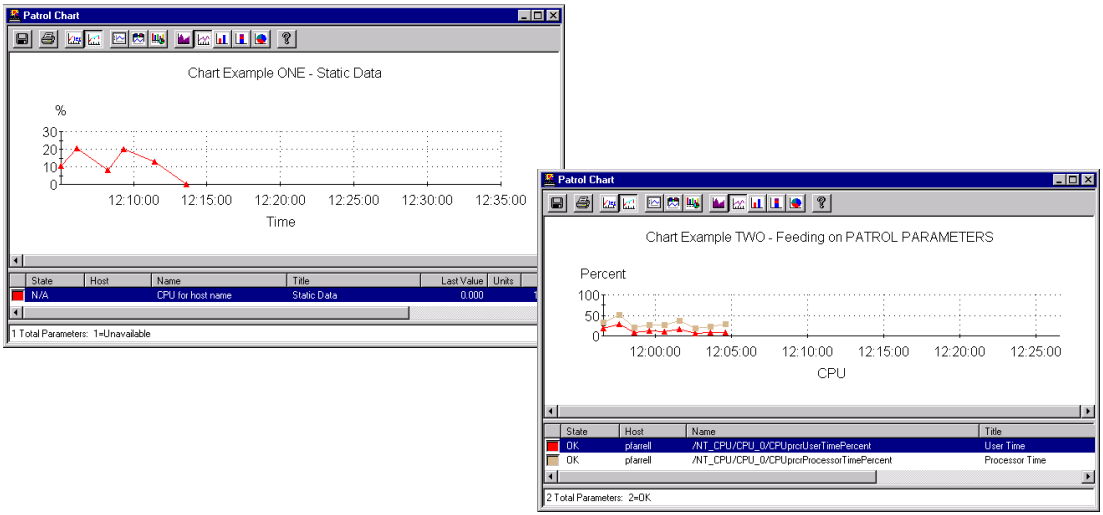
The following example defines a chart with static data, destroys the static chart, and then creates a dynamic chart:

```
# define chart one
ct = "ChartTitle: Chart Example ONE - Static Data\n";
x = "XAxisLabel: Time\n";
y = "YAxisLabel: %\n";
d1 = "StaticData2D:CPU for host name,836586286 10.5, 836586386 20.4, 836586486 7.90, 836586586 20,
836586686 13, 836586786 0\n";
# build the GDF string
```

BMC Software, Inc., Confidential and Proprietary Information

```
gdf_string = ct.x.y.d1;
#display chart one
id = chart(CHART_LOAD, gdf_string);
chart(CHART_PRINT, "", id);
sleep(10);
chart(CHART_DESTROY, "", id);
# define chart two
ct = "ChartTitle: Chart Example TWO - Feeding on PATROL PARAMETERS\n";
x = "XAxisLabel: CPU\n";
y = "YAxisLabel: Percent\n";
# pfarrell is my PATROLAGENT machine
host_name = get("hostname");
d1 = "PatrolConsole:".host_name.",/NT_CPU/CPU_0/CPUprcrUserTimePercent\n";
d2 = "PatrolConsole:".host_name.",/NT_CPU/CPU_0/CPUprcrProcessorTimePercent\n";
# build the GDF string
gdf_string = ct.x.y.d1.d2;
# display chart two
id = chart(CHART_LOAD,gdf_string);
chart(CHART_PRINT, "", id);
sleep(10);
chart(CHART_DESTROY, "", id);
```

This example creates the following charts:



close()

Close a file or process channel

Format

`close(`*`channel`*`,[`*`flags`*`])`

Parameters

Parameter	Definition
channel	the process or file I/O channel name (shared channels) or number (local channels) that is to be closed
flags	optional bit flags used to control *close* execution The following bits are used: • Bit 1 = 1 indicates that any system process associated with the channel (that is, the PSL `fopen()` or `popen()` function) should be killed while closing the channel • Bit 2 = 1 indicates that the channel should be closed even if another PSL process is blocked waiting for a `read()`, `readln()`, or `write()` function. Bit 2 applies only to global channels and is ignored by local channels. • **Default** Bits 1 and 2 are both zero.

Description

The `close()` function closes a channel to a process or command previously created by a `fopen()` or `popen()` call.

When *flags* are not specified, the default is zero.

BMC Software, Inc., Confidential and Proprietary Information

When bit 1 = 0, the `close()` function does not kill any processes spawned as a result of the `fopen()` or `popen()`, and these processes are allowed to continue. This feature of `close()` allows you to open a channel to a PSL process, send additional data, and close the channel while allowing the process to complete.

When bit 2 = 1, the `close()` function will close the channel even if another PSL process is blocked pending an I/O request on that channel. When blocking occurs, `close()` causes the blocked function to wake and receive an error return and `errno` from the process to which the channel was opened.

The `close()` function returns the NULL string if the closure was successful and –1 with the PSL variable `errno` set if the closure was unsuccessful. The `close()` function fails when bit 2 = 0 and *channel* is a global channel with at least one blocked PSL process.

Example

This example closes the channel that is represented by *chan*:

```
close(chan);
close(chan,1);   # bit 1 set
close(chan,2);   # bit 2 set (2 is binary 10)
close(chan,3);   # both bits set (3 is binary 11)
```

BMC Software, Inc., Confidential and Proprietary Information

code_cvt()

Convert a string from one codeset to another

Format

```
code_cvt(source_cs, dest_cs, text)
```

Parameters

Parameter	Definition
source_cs	codeset of *text* **Valid Values** 0, "", or any supported codeset name (Table 1-4 on page 1-57) If 0 or "" is specified, this function uses the codeset of the CTYPE locale category.
dest_cs	codeset to which you want to convert *text* **Valid Values** Same as *source_cs*.
text	text string that you want to convert

Description

The code_cvt() function converts *text* from the codeset specified by *source_cs* to the *dest_cs* codeset and returns *text*. You can convert only between two codesets that represent the same language. For example, you can convert between CP932 and eucJP because both codesets represent the Japanese language.

Table 1-4 shows the codesets that you can use with this function.

Table 1-4 Supported Codeset Names

Codeset Name	Language
C	English
CP932	Japanese
eucJP	Japanese
SJIS	Japanese
eucKR	Korean
gb	Simplified Chinese
big5	Traditional Chinese
eucTW	Traditional Chinese

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

If the code_cvt() function fails, it returns an empty string and sets the PSL *errno* variable to one of the values in the following table:

errno Value	Description of Failure
135	E_PSL_CODECVT_INVALID_MB *text* contains invalid characters. This function replaces each invalid character with a question mark and returns *text.*
136	E_PSL_CODECVT_NO_CONV No support for conversion between *source_cs* and *dest_cs*.
137	E_PSL_CODECVT_SAME *Source_cs* and *dest_cs* are the same codeset.

Note

If *text* is an empty string, code_cvt() returns an empty string, but it does not assign a value to *errno*.

Example

In this example, a user-defined function runs the code_cvt() function, verifies that the conversion was successful, and displays the result.

```
function use_code_cvt(id, from, to, data, err)
{
   ret = code_cvt(from, to, data);
   if (ret == 0 || errno != err)
   {
       printf("Conversion %d failed. ret = 0, errno = %d,
       expect_err = %d\n",id, errno, err);
   }
   else
   {
       save_err = errno;
       printf("ID %d, conversion is complete -- \n", id);
       printf("  [from] %s\n", data);
       printf("    [to] %s<<<errno=%d>>>\n", ret, save_err);
   }
}
```

BMC Software, Inc., Confidential and Proprietary Information

cond_signal()

Signal a process that is blocked on a condition wait

Format

`cond_signal(`*`variable`*`,[`*`all`*`])`

Parameters

Parameter	Definition
variable	name of the variable that will unblock a process blocked by the `cond_wait()` function
all	non-NULL value that directs the `cond_signal()` function to unblock all PSL processes that are blocked waiting for *variable*

Description

The `cond_signal()` function can signal another PSL process that is currently blocked for a `cond_wait()` function on *variable*. If *all* is specified and is not the NULL string, the `cond_signal()` function will wake all PSL processes that are blocked on *variable*. If no processes are blocked on *variable*, the `cond_signal()` function has no effect. The `cond_signal()` function can never block and always returns the NULL string.

Example

The following cond_signal() function signals all processes waiting for CV1 to unblock after the process PASSWD_LCK unlocks:

```
unlock("PASSWD_LCK");
cond_signal("CV1");
```

BMC Software, Inc., Confidential and Proprietary Information

cond_wait()

Block a process until a condition signal is received

Format

`cond_wait(`*`variable`*`,`*`lockname`*`,[`*`timeout`*`])`

Parameters

Parameter	Definition
variable	name of the variable that will end the `cond_wait()` condition The *variable* is issued by the `cond_signal()` function.
lockname	the name of the lock the `cond_wait()` function should attempt to acquire when it receives the correct unblocking *variable* in the `cond_signal()` function If *lockname* is the NULL string, the `cond_wait()` function will not attempt to acquire a lock after receiving *variable*.
timeout	optional number of seconds to wait for the receipt of *variable* before unblocking and releasing *lockname* **Valid Values** • *timeout* > 0 specifies the timeout value in seconds • *timeout* < 0 specifies an infinite timeout; only the receipt of *variable* can unblock the process Specifying *timeout* = 0 is not permitted and will result in a PSL run-time error message. **Default** infinite timeout

BMC Software, Inc., Confidential and Proprietary Information

Description

The `cond_wait()` function blocks the current PSL process until either *variable* is received or until *timeout* expires. If the PSL process holds *lockname* when the `cond_wait()` function is issued, the `cond_wait()` function releases the lock. When the `cond_wait()` function receives *variable*, the `cond_wait()` function immediately attempts to acquire *lockname*.

If the `cond_wait()` function returns a 1, it will always hold an exclusive lock on *lockname*. If the `cond_wait()` function fails, it will not hold any form of lock on *lockname* when it returns. If a timeout occurs, the `cond_wait()` function returns a failure value of “0,” sets the PSL
`errno` = 95 (E_PSL_TIMEOUT) and will not hold any lock on *lockname*.

condition_variable

The parameter *variable* is the name of the condition variable that the `cond_wait()` function waits to have signaled by the `cond_signal()` function. Condition variable names have global scope analogous to locks and shared channels. None of these different global scopes interfere with one another. You can use the same name without conflict for a lock, a shared channel, and a condition variable.

lockname

On entry to the `cond_wait()` function, the process releases the lock *lockname* and blocks waiting to be signaled. The *lockname* should usually be an exclusive lock held by this process; otherwise, run-time error messages may occur (although the `cond_wait()` function will still try to go ahead and wait for a signal anyway). The `cond_wait()` function will always block waiting for the `cond_signal()` function or for *timeout*.

BMC Software, Inc., Confidential and Proprietary Information

When another PSL process performs a `cond_signal()` function that wakes this PSL process, the `cond_wait()` function call will attempt to gain an exclusive lock (if a lock is requested, that is, if *lockname* is not the empty string) and either return immediately with the lock or join the queue waiting for an exclusive lock on *lockname*.

It is common style to supply *lockname* because condition variables are almost always shielded by locks. In the `cond_wait()` function, *lockname* must be supplied as the NULL string rather than omitted to force the PSL coder to consider whether a lock is needed. The required *lockname* reduces the number of errors caused by not using a lock when one is needed.

timeout

The behavior of *timeout* is unchanged regardless of whether the `cond_wait()` function is waiting for a `cond_signal()` function or waiting to acquire *lockname*. If *variable* or *lockname* is destroyed before the `cond_wait()` function is complete, the `cond_wait()` function returns 0 and sets the PSL `errno` value but will not hold any lock on *lockname*. The lock *lockname* can be the NULL string, in which case *variable* is considered to have no associated lock; and the `cond_wait()` function will return success immediately upon being signaled without waiting for any lock.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following cond_wait() function waits on the signal CV1 from PASSWD_LCK before attempting to get the lock:

```
lock("PASSWD_LCK");
while ((ret = cond_wait("CV1","PASSWD_LCK",180)) != 1) {
   if ((ret == 0) && (errno == 95)) {
       last;
   }
   print("Still waiting....\n");
}
if (errno == 95) {
   print("Timeout before lock was aquired\n");
} else {
   print("Got the lock\n");
}
```

BMC Software, Inc., Confidential and Proprietary Information

console_type()

Return 1 if the console is a PATROL Developer Console

Format

```
console_type()
```

Parameters

This function has no parameters.

Description

The `console_type()` function examines the console for which the PSL script is running and returns a 1 if the console is a PATROL Developer Console, or returns a 0 otherwise.

Example

The following console_type() function prints the type of console from which it is running:

```
if (console_type()) {
   print("Developer Console\n");
} else {
   print("Operator Console\n");
}
```

BMC Software, Inc., Confidential and Proprietary Information

convert_base()

Convert a number from one base to another

Format

`convert_base(`*`number`*`,`*`from_base`*`,`*`to_base`*`,[`*`sign`*`])`

Parameters

Parameter	Definition
number	the number that is converted from *from_base* to *to_base* **Valid Values** (in base 10) Signed –2147483647 to 2147483647 Unsigned 0 to 4294967295 On some 64-bit machines, these numbers can be larger.
from_base	the base from which *number* is converted **Valid Values** 2–36
to_base	the base to which *number* is converted **Valid Values** 2–36
sign	an optional flag to specify if *number* is signed If this value is omitted, the default is signed. A non-zero value indicates that the value is unsigned.

Description

The convert_base() function converts a numeric value, *number*, from one base, *from_base*, to another base, *to_base*.

The convert_base`()` function returns an empty string and sets the following PSL errno values under the following conditions:

errno Value	Description
68	E_PSL_EDOM one or both *base* values are out of range
69	E_PSL_ERANGE the *number* value is out of range

If any part of *number* is invalid for *from_base*, convert_base() stops the conversion and returns any part of *number* that it was able to read. For example, the command `convert_base("78", 8, 10);` returns 7 because 8 is invalid for base 8. If you enter text for *number*, you might find it harder to notice when you have entered an invalid *number* for *from_base*. For example, the command `convert_base("az", 16, 10);` returns 10 because, in base 16, “a” equals 10 and “z” is invalid.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL example converts a base 10 number supplied by the user to a base specified by the user. This function uses the convert_base() function for the base conversion and the response() function to create a GUI for the user to enter the desired values:

```
# Response function to input number and desired base
string=response("Base Conversion",-1,
   ["h=100","w=100","b=Patrol","e=1","o=Accept","c=Cancel ",
   "N=0","A=0","B=0"],
   [R_COLUMN, 4],
   [R_LABEL_CENTER, "Enter a base 10 number,\n
   and select a base to convert it to:"],
   [4],[R_LABEL, "              "],
   [R_TEXT_FIELD_LABEL, "Number", "\n", "e=1"],
   [R_ROW, 2],
   [R_LABEL, "              "],
   [R_CLICKER, "Base", 2,36,2]
);
# Obtain status, number, and desired base from response function return
   stat=trim(nthlinef(string,"1"),"\n");
   numb=trim(nthlinef(string,"2"),"\n");
   base=trim(nthlinef(string,"3"),"\n");
# Verify the return status if the dialog was CANCELLED (0), exit
   if (stat==0) {
   exit;
# Verify number is numeric and within range if not display error message
   } elsif (isnumber(numb)==0 || numb>2147483647 || numb<-2147483647){
   response("Error",-1,
   ["h=100","w=100","b=Patrol","e=1","o=OK","c=Cancel ","N=1","A=0","B=0"],
   [R_COLUMN, 3],[4],[R_LABEL, "              "],
   [R_LABEL_CENTER, "The value entered\n\n"."(".numb.")"."\n\n
   is not a valid number, or it is out of range!"]
   );
exit;
# If input is acceptable, compute new value and display output dialog
   } else {
# Use the convert_base() function to convert numb from base 10 to base
   new=convert_base(numb,"10",base);
   response("Conversion",-1,["h=100","w=100","b=Patrol","e=1","o=OK",
   "c=Cancel ","N=1","A=0","B=0"],
   [R_COLUMN, 3],[4],[R_LABEL, "              "],
   [R_LABEL_CENTER, numb." converted to base ".base." = ".new]
   );
}
```

BMC Software, Inc., Confidential and Proprietary Information

convert_date()

Convert the Gregorian date and time to the number of seconds that have elapsed since 00:00:00 GMT January 1, 1970

Format

```
convert_date(day date month year time [timezone],
      [timezone])
convert_date(day month date time [timezone] year,
      [timezone])
convert_date(day month date time year,[timezone])
```

Parameters

Parameter	Definition
day	three-character string representing the day of the week **Valid Values** Sun Mon Tue Wed Thu Fri Sat
date	integer day of the month **Valid Values** 1–31
month	three-character string representing the month **Valid Values** Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
year	integer year **Valid Values** 1902–2037

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
time	eight-character string in the form *hh:mm:ss* representing the time **Valid Values** *hh* hours (00–23) *mm* minutes (00–59) *ss* seconds (00–59)
timezone	optional time zone for which the conversion is being performed **Valid Values** • A through I, K through Z—military time zones • ADT—Atlantic Daylight • AHST—Alaska-Hawaii Standard • AST—Atlantic Standard • AT—Azores • BST—British Summer • BT—Baghdad, USSR Zone 2 • CADT—Central Australian Daylight • CAST—Central Australian Standard • CAT—Central Alaska • CCT—China Coast, USSR Zone 7 • CDT— Central Daylight • CET— Central European • CST— Central Standard • EADT— Eastern Australian Daylight • EAST— Eastern Australian Standard • EDT— Eastern Daylight • EET— Eastern Europe, USSR Zone 1 • EST— Eastern Standard • FST— French Summer • FWT— French Winter • GMT— Greenwich Mean • GST— Guam Standard, USSR Zone 9 • HDT— Hawaii Daylight • HST— Hawaii Standard • IDLE— International Date Line East • IDLW—International Date Line West • IST— Indian Standard • IT— Iran • JST— Japan Standard, USSR Zone 8 • JT— Java (3pm in Cronusland!) • MDT— Mountain Daylight • MEST— Middle European Summer • MET— Middle European • MEWT— Middle European Winter

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
timezone **(continued)**	• IT— Iran • JST— Japan Standard, USSR Zone 8 • JT— Java (3pm in Cronusland!) • MDT— Mountain Daylight • MEST— Middle European Summer • MET— Middle European • MEWT— Middle European Winter • MST— Mountain Standard • NDT— Newfoundland Daylight • NFT— Newfoundland • NST— Newfoundland Standard • NT——Nome • NZDT— New Zealand Daylight • NZST— New Zealand Standard • NZT— New Zealand • PDT— Pacific Daylight • PST— Pacific Standard • SST— Swedish Summer • SWT— Swedish Winter • TZ1— See User-Defined Time Zone on next page. • TZ2— See User-Defined Time Zone on next page. • UT or UTC— Universal (Coordinated) • WADT— West Australian Daylight • WAST— West Australian Standard • WAT— West Africa • WET— Western European • YDT— Yukon Daylight • YST— Yukon Standard • ZP4— USSR Zone 3 • ZP5— USSR Zone 4 • ZP6— USSR Zone 5

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition

timezone
(continued)
	Relative Time Zones
	The time zone can also be specified as a relative value to one of the supported time zones. The following example would specify Greenwich Mean Time plus eight hours:
	`"GMT+0800"`
	
	User-Defined Time Zones
	You can assign values to the TZ1 and TZ2 time zones by using the AgentSetup/PATROLTZ variable. This variable uses the format `REPLACE="TZ1/`*`offset1`*`,TZ2/`*`offset2`*`"` where *offset1* and *offset2* represent the difference (in minutes) between GMT and the user-defined time zone. For example, the following syntax would mean that the first time zone is three hours ahead of GMT while the second time zone is one hour behind GMT:
	`REPLACE="TZ1/+180,TZ2/-60"`
	
	Default
	computer system local time zone

Description

The convert_date() function returns an integer that is the number of seconds past 00:00:00 GMT January 1, 1970 of the specified date and time string. The convert_date() command is capable of parsing any of three date and time string formats:

- the RFC-822 format (*day, date month year time timezone*)
- the Unix format (*day month date time timezone year*)
- the PSL date() function format (*day month date time year*)

The convert_date() function requires the *timezone* parameter in order to correctly convert the date and time. If *timezone* is not specified, either in the date and time string (RFC-822 and Unix formats) or as a separate parameter (all formats), the convert_date() function uses the time zone defined for the machine on which it is executing.

BMC Software, Inc., Confidential and Proprietary Information

If the convert_date() function encounters an error, it returns a zero and sets the PSL errno variable as follows:

errno Value	Description
105	E_PSL_BAD_DATE_STRING indicates that the convert_date() function could not parse the date and time string
106	E_PSL_BAD_TIMEZONE indicates that the convert_date() function did not recognize the specified *timezone*
107	E_PSL_CANNOT_DETERMINE_TIMEZONE indicates that the convert_date() function is using *timezone* = GMT because no *timezone* was supplied and the time zone used by the computer system could not be determined

BMC Software, Inc., Confidential and Proprietary Information

Example

The following convert_date() function calls illustrate the different ways to specify the *timezone* parameter:

```
# Specify timezone as part of the date and time string
seconds = convert_date("Fri Mar  8 11:39:07 CST 1996");
print("Converted time is ",seconds,"\n");

# Specify timezone as a separate parameter
seconds = convert_date("Fri Mar  8 11:39:07 1996","CST");
print("Converted time is ",seconds,"\n");

# Use time zone defined in the operating system
seconds = convert_date("Fri Mar  8 11:39:07 1996");
print("Converted time is ",seconds,"\n");
```

In the case where the operating system uses the CST time zone, the example produces the following output:

```
Converted time is 826306747
Converted time is 826306747
Converted time is 826306747
```

BMC Software, Inc., Confidential and Proprietary Information

convert_locale_date()

Convert international date and time formats to the number of seconds that have elapsed since 00:00:00 GMT January 1, 1970

Format

`convert_locale_date(`*`date_string`*`,[`*`format`*`])`

Parameters

Parameter	Definition
date_string	a string representing a date
format	one or more symbols that describe the format of *date_string* See Table 1-5 for a list of format symbols. Each symbol is composed of a percent (%) sign followed by one or two characters. If you use more than one symbol, separate each symbol by a space. Default day month date time year (Fri Jan 2 08:00:00 1970)

Description

The convert_locale_date() function converts a date string to the number of seconds that have elapsed since 00:00:00 GMT January 1, 1970.

The convert_locale_date() function serves the same purpose as convert_date(). With convert_locale_date(), however, you can use the format symbols listed in Table 1-5 on page 1-75 to define the format of the *date_string* parameter. These format symbols provide the flexibility required to enter international date and time formats.

BMC Software, Inc., Confidential and Proprietary Information

If you do not use the *format* parameter, convert_locale_date() accepts the format used by the PSL date() function (day month date time year).

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Table 1-5 Format Symbols for convert_locale_date() (Part 1 of 3)

Format Symbol	Description
%%	with this symbol, you can use a percent sign (%) in the format of a date string
%a	locale's name of the day of week; either the abbreviated or full name may be specified.
%A	same as %a
%b	locale's name of the month; either the abbreviated or full name may be specified.
%B	same as %b
%c	locale's appropriate date and time representation
%C	century number [0,99]; leading zero is permitted but not required (no change to tm)
%d	day of month [1,31]; leading zero is permitted but not required
%D	date as %m/%d/%y
%e	same as %d
%h	same as %b
%H	hour (24-hour clock) [0,23]; leading zero is permitted but not required
%I	hour (12-hour clock) [1,12]; leading zero is permitted but not required
%j	day of year [1,366]; leading zeros are permitted but not required
%m	month as a decimal number [1,12]; leading zero is permitted but not required
%M	minute [0,59]; leading zero is permitted but not required
%n	any white space
%p	locale's equivalent of either a.m. or p.m.
%r	appropriate time representation in 12-hour clock format with %p

BMC Software, Inc., Confidential and Proprietary Information

Table 1-5 Format Symbols for convert_locale_date() (Part 2 of 3)

Format Symbol	Description
%R	time as %H:%M
%S	seconds [0,61]; leading zero is permitted but not required
%t	any white space
%T	time as %H:%M:%S
%U	week of the year as a decimal number [0,53], with Sunday as the first day of the week; leading zeros are permitted but not required (no change to tm)
%w	day of week as a decimal number [0,6], with 0 representing Sunday
%W	week of the year as a decimal number [0,53], with Monday as the first day of the week; leading zero is permitted but not required (No change to tm)
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year within the century [0,99]; leading zero is permitted but not required
%Y	year, including the century (for example 1993)
%Z	name of time zone, or no characters if no information of the time zone exists
%Ec	locale's alternative appropriate date and time representation
%Ex	locale's alternative date representation
%EX	locale's alternative time representation
%Od	day of the month using the locale's alternative numeric symbols
%Oe	same as %Od
%OH	hour (24-hour clock) using the locale's alternative numeric symbols
%OI	hour (12-hour clock) using the locale's alternative numeric symbols
%Om	month using the locale's alternative numeric symbols
%OM	minutes using the locale's alternative numeric symbols
%OS	seconds using the locale's alternative numeric symbols
%OU	week of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols
%Ow	day of week (Sunday=0) using the locale's alternative numeric symbols

BMC Software, Inc., Confidential and Proprietary Information

Table 1-5 Format Symbols for convert_locale_date() (Part 3 of 3)

Format Symbol	Description
%OW	week of the year (Monday as the first day of the week) using the locale's alternative numeric symbols
%Oy	year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols

Example

The following example demonstrates the convert_locale_date() function:

```
set_locale("TIME", "C");
ret = convert_locale_date("Fri Feb 2 08:00:00 1970");
printf("\nConvert date results: locale = C,
    date = Fri Jan 2 08:00:00 1970, and
    converted date = %s\n", ret);

ret=convert_locale_date("13:11:00 2 2001","%T %j %Y");
printf("\nConvert date results: locale = C,
    date = 1:11 PM Jan 2, 2001, and
    converted date = %s\n", ret);
```

The following is the output from the previous example.

```
Convert date results: locale = C,
date = Fri Jan 2 08:00:00 1970, and
converted date = 136800

Convert date results: locale = C,
date = 1:11 PM Jan 2, 2001, and
converted date = 978289860
```

BMC Software, Inc., Confidential and Proprietary Information

cos()

Return the cosine of the argument

Format

`cos(`*`radians`*`)`

Parameter

Parameter	Definition
radians	arc length in radians whose cosine is to be determined **Valid Values** $-\infty \leq$ *radians* $\leq \infty$

Description

The `cos()` function returns the cosine of *radians*. The output range for the `cos()` function is $-1 \leq$ `cos()` ≤ 1.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the cos() function to draw a crude graph of a cosine waveform over a half period:

```
print("Cosine Waveform:\n\n");
print("     -1 -0.5   0   0.5   1\n");
print("     +----+----+----+----+\n");
i = 0.0; while (i <= 3.2) {
   printf("%4.2f ",i);
   cosine = cos(i);
   plot = int(10 * cosine) + 10;
   if (plot < 10) {
      before = "";
      j = 1; while (j <= plot) {
         before = before . " ";
         j++;
      }
      after = "";
      j = plot; while (j < 9) {
         after = after . " ";
         j++;
      }
      print(before,"*",after,"|");
      printf("            cosine = %+8.6f\n",cosine);
   } elsif (plot == 10) {
      print("          *");
      printf("            cosine = %+8.6f\n",cosine);
   } else {
      print("          |");
      before = "";
      j = 11; while (j < plot) {
         before = before . " ";
         j++;
      }
      print(before,"*");
      after = "";
      j = plot + 1; while (j <= 22) {
         after = after . " ";
         j++;
      }
      print(after);
      printf("cosine = %+8.6f\n",cosine);
   }
   i = i + 0.1;
}
```

BMC Software, Inc., Confidential and Proprietary Information

The example produces the following output:

```
Cosine Waveform:

     -1 -0.5   0   0.5   1
     +----+----+----+----+
0.00           |          *  cosine = +1.000000
0.10           |         *   cosine = +0.995004
0.20           |         *   cosine = +0.980067
0.30           |         *   cosine = +0.955336
0.40           |         *   cosine = +0.921061
0.50           |        *    cosine = +0.877583
0.60           |        *    cosine = +0.825336
0.70           |       *     cosine = +0.764842
0.80           |      *      cosine = +0.696707
0.90           |      *      cosine = +0.621610
1.00           |     *       cosine = +0.540302
1.10           |    *        cosine = +0.453596
1.20           |   *         cosine = +0.362358
1.30           |  *          cosine = +0.267499
1.40           |*            cosine = +0.169967
1.50           *             cosine = +0.070737
1.60          *|             cosine = -0.029200
1.70         * |             cosine = -0.128844
1.80        *  |             cosine = -0.227202
1.90       *   |             cosine = -0.323290
2.00      *    |             cosine = -0.416147
2.10     *     |             cosine = -0.504846
2.20     *     |             cosine = -0.588501
2.30    *      |             cosine = -0.666276
2.40   *       |             cosine = -0.737394
2.50  *        |             cosine = -0.801144
2.60  *        |             cosine = -0.856889
2.70 *         |             cosine = -0.904072
2.80 *         |             cosine = -0.942222
2.90 *         |             cosine = -0.970958
3.00 *         |             cosine = -0.989992
3.10 *         |             cosine = -0.999135
3.20 *         |             cosine = -0.998295
```

cosh()

Return the hyperbolic cosine of the argument

Format

`cosh(`*`argument`*`)`

Parameter

Parameter	Definition
argument	numeric value whose hyperbolic cosine is to be determined **Valid Values** $-\infty \leq$ *argument* $\leq \infty$

Description

The `cosh()` function returns the hyperbolic cosine of *argument*. The hyperbolic cosine is defined by the expression:

$$\cosh(x) = (e^{x} + e^{-x})/2$$

where e is the base for the natural logarithms (e =2.71828 . . .). The output range for the `cosh()` function is $1 \leq$ `cosh()` $\leq \infty$.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the `cosh()` function to numerically check the hyperbolic cosine definition:

```
function main() {
   print("Test cosh(x) = 0.5 * (exp(x) + exp(-x))\n");
   print("over the interval from -5 to 5\n\n");
   print("    x         cosh(x)   0.5 * (exp(x) + exp(-x))\n");
   print("----------  ----------  -----------------------\n");
   i = - 5;
   while (i <= 5) {
      coshdef = 0.5 * (exp(i) + exp(-i));
      printf("%+10.6f  %+10.6f       %+10.6f\n",i,cosh(i),coshdef);
      i++;
   }
}
```

BMC Software, Inc., Confidential and Proprietary Information

create()

Create a PATROL object

Format

`create(`*`object`*`,[`*`label`*`],[`*`state`*`],[`*`description`*`],[`*`parent`*`])`

Parameters

Parameter	Definition
object	an alphanumeric identifier for the object. The *object* can be either a string (enclosed in double quotation marks) or a variable containing a string and must be unique within the current application's name space. The *object* parameter is validated using the following rules: • *object* is composed of name segments separated by a slash (/) • *object* may start with a slash (/), but not end with a slash (/) • *object* may not contain empty segments • *object* may not contain unprintable characters • *object* may not contain whitespace other than the space character • *object* may contain spaces within a segment but not at the beginning or end of a segment
label	optional text string that is used for object identification and icon specification If no icon is specified in the label, then the label appears below the object's icon. If an icon is specified in the label, then the label is split into two components—the label string that will appear under the object's icon, and the name of the icon to be used. **Default** *object*

Parameter	Definition
state	optional state of the object assigned at its creation **Valid Values** • OFFLINE • OK • WARN • ALARM **Default** NONE **Note:** Typically you should create an object in the OK state. Using any other state causes PATROL to suspend the parameters that the object may have. Also the PATROL Console does not display the icon of the object until the object has a valid state.
description	optional text string that can be used to introduce the newly created object The text string must be enclosed in double quotation marks. The text string is recorded in the Event Description of the State Change Event in the Event Manager. **Default** NULL string
parent	optional text string that specifies the parent to which the created object will belong If no parent is specified, object will not have a parent. If parent is `"/"`, *object* is displayed in the PATROL Main window.

Description

The `create()` function creates the PATROL application instance with the identifier *object*.

If *label* is specified, it is the label that the PATROL Console displays beneath the object's icon. The default label is the *object* name.

The format of the label string is as follows:

`"instance_name"  |  ":instance_name:icon_base_name"`

In both cases, the *instance_name* will be used as the label for the icon. The *icon_base_name* specifies the two icons which will be used to show the states of the instance. In the Console, the two icons which are used will be `icon_base_name_ok` and `icon_base_name_warn`. The suffix and format are defined by the Console according to the PSL rules, so you do not have to specify the file extension in the label string. If the `_ok` or `_warn` icon, or both, is not found, the console will use the icon defined in the APPLICATION class.

Note

The Unix Console uses .xpm icon files, and the NT Console uses .bmp icon files.

If *parent* is specified, *parent* will be the parent instance of *object*. Thus,

- The container for *parent* on the PATROL Console will contain *object*.
- When *parent* is destroyed, *object* will also be destroyed.
- If *object* is allowed to propagate its state, and is in ALARM, the combined state of *parent* will also be in ALARM.

The `create()` function returns *object* if it creates the instance or the NULL string if the instance already exists or an error occurs. If the *object* parameter syntax is invalid, the `create()` function sets the `errno` variable to 96 indicating `E_PSL_BAD_FUNCTION_PARAMETER`.

Since the PATROL Agent and Console do not share a single hierarchy of application, instance, and parameter, it is possible for a new application instance icon to be missing on the console. When an object is created with the `create()` function and a parent object is specified, the object ID of the parent object is stored in the application instance structure on the agent. This guarantees the existence of the parent object in the agent namespace, but not necessarily on the console.

BMC Software, Inc., Confidential and Proprietary Information

When the application instance is sent to the console, the agent includes the parent object ID, which allows the console to determine where the icon should be located.

The icon will be misplaced or not displayed in the following circumstances:

- The parent object exists in the agent namespace but does not have an icon in the console. The parent object won't have an icon in the following circumstances:

  — The parent object is in the VOID state.
  — The KM to which the parent object belongs is not loaded on the console.
  — The console has not asked the agent for the status of the parent object yet.

- The parent object has been destroyed.

  — The KM destroyed the instance via PSL.
  — A developer is downloading a new version of the KM to the agent.

When using the `create()` function, it is good practice to check if a parent object exists and is not in a VOID state before the object is created.

Examples

The following examples highlight the usage of the `create()` function.

Create an object with a label, state, and description:

```
# This object is created in OK condition and thus displayed.
create(name, "martin", OK,"User logged in");
```

Create an object with no optional parameters:

```
# This object is created but not displayed.
create("Secret_object");
```

Create an object with an Initial “OK” state:

```
# This object is created with the default label.
create(name, "",OK);
```

Create an object with a label:

```
# This example creates the root instance of the FILESYSTEM
# application. The label "root" appears under the object’s
# icon.
create("/FILESYSTEM/root", "root", OK);
```

Create an object with a label and specified icons:

```
# This example creates the root instance of the FILESYSTEM
# application with the label "root"appearing under the
# object’s icon. The icons that will be
# used for the object are root_icon_ok.xpm and
# root_icon_warn.xpm (for the 3.2 Unix Console).
create("/FILESYSTEM/root", ":root:root_icon", OK);
```

BMC Software, Inc., Confidential and Proprietary Information

date()

Return the date and time as a 24-character string

Format

```
date()
```

Parameters

This function has no parameters.

Description

The `date()` function returns the current date and time as a 24-character string in the format:

```
Sun Sep 16 01:03:52 1973
```

The `date()` function is equivalent to the C library `ctime(3)` function. The `date()` function is also equivalent to the PSL statement:

```
asctime(time());
```

Example

The following example prints the current date and time to the PATROL Console:

```
today = date();
print("The date is ",today);
```

BMC Software, Inc., Confidential and Proprietary Information

dcget_text()

Get message text by defining locale domain and category

Format

`dcget_text(`*`domain_name, message_ID, category`*`)`

Parameter

Parameter	Definition
domain_name	file name for message catalog Use only characters that are permitted in file names.
message_ID	text string that identifies a message
category	locale category Name of the locale category also determines the name of the directory that contains the domain file. **Valid values** CTYPE, MESSAGES, TIME, and CODECVT

Description

The dcget_text() function returns a message that has a text string as the message ID. The get_text() and dget_text() functions serve the same purpose, but dcget_text() requires that you specify *domain_name* and *category*. These parameters determine the name of the catalog file and the directory path according to the following format:

PATROL_HOME/**lib**/**nls**/**locale**/*locale_name*/**ML**_*category*/*domain_name*.**bo**

BMC Software, Inc., Confidential and Proprietary Information

The dcget_text() function returns *message_ID* when it cannot find a message that matches the specified *message_ID*, *domain_name* and *category*. It also returns *message_ID* when the current locale is the same as the CTYPE locale.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Example

The following example demonstrates the dcget_text() function.

```
function display_dcget_text(domain, msg_id, category)
{
   ret = dcget_text(domain, msg_id, category);
   printf("Message text = %s", ret);
}

display_dcget_text("test_msg", "hello", "CODECVT");
```

debugger()

Suspend process pending an attach command from the PSL debugger

Format

```
debugger()
```

Parameters

This function has no parameters.

Description

The PSL `debugger()` function suspends the current PSL process waiting for an attach command from the PSL debugger. The `debugger()` function complements options within the PSL debugger that suspend PSL processes. The `debugger()` function offers a low-level method of stopping a PSL process for debugging before it begins.

Although modifying PSL source code to debug a particular script may not be convenient, the `debugger()` function provides a generalized method whereby all PSL code can be debugged, even if it would normally be executed before the console user has time to raise a PSL debugger window. The suspension of the PSL process occurs regardless of whether a PSL debugger window is active on the console.

The only way to restart a PSL function suspended through the `debugger()` function is through the PSL debugger. Thus, the `debugger()` function should only be used in non-production versions of Knowledge Modules.

If the PSL process is already being processed by the PSL debugger, a call to the `debugger()` function call has no effect. The `debugger()` function always returns the NULL string.

BMC Software, Inc., Confidential and Proprietary Information

desktop()

Send a PATROL link command to the PATROL Console for execution

Format

`desktop(`*`command`*`,[`*`application`*`])`

Parameters

Parameter	Definition
command	a PATROL link command for the console to execute
application	optional name of a valid application **Default** The *command* is sent to all consoles that are interested in the PSL process.

Description

The `desktop()` function sends a specified PATROL link *command* to the console for execution. If *application* is not specified, the `desktop()` function sends *command* to all consoles that are interested in the PSL process. If *application* is specified and is a valid application (that is, the application exists), then the output of *command* is sent to all consoles that are interested in that application.

Example

The following is an example of the `desktop()` function:

```
# Open the FSCapacity parameter of the root instance of the FILESYSTEM
# application for the nebula agent.
desktop("open nebula:/FILESYSTEM/root/FSCapacity");
# Open it only on those systems that are monitoring the ORACLE application.
desktop("open nebula:/FILESYSTEM/root/FSCapacity","ORACLE");
```

BMC Software, Inc., Confidential and Proprietary Information

destroy()

Destroy a PATROL object

Format

`destroy(`*`object`*`,[`*`description`*`])`

Parameters

Parameter	Definition
object	the alphanumeric identifier for the object The *object* is assigned when the object is created.
description	optional text string that can be used to explain why the object was destroyed The text string is recorded in the Description of the State Change Event in the Event Manager. **Default** NULL string.

Description

The `destroy()` function deletes the PATROL application instance *object* and returns TRUE on success and FALSE on error.

Example

The following example destroys an object represented by the variable *name*:

```
# destroy object whose name is in variable ‘name’
destroy(name);
```

BMC Software, Inc., Confidential and Proprietary Information

destroy_lock()

Destroy a PSL process lock and wake all processes blocked on the lock

Format

`destroy_lock(`*`lockname`*`)`

Parameter

Parameter	Definition
lockname	name of the lock that is destroyed

Description

The `destroy_lock()` function destroys *lockname* that was previously granted to a PSL process and wakes all processes that are blocked on it. (Processes become blocked on a lock through either the lock() or cond_wait() function calls.) The destroy_lock() function returns 1 if it successfully finds and destroys *lockname*, and 0 if it does not.

Note

The lock() and cond_wait() functions set the PSL variable errno = 114 (E_PSL_LOCK_DESTROYED) if the lock upon which they are blocked is destroyed.

Example

The following destroy_lock() function destroys the lock FOOBAR:

```
status = destroy_lock("FOOBAR");
```

BMC Software, Inc., Confidential and Proprietary Information

dget_text()

Get message text by defining locale domain

Format

`dget_text(`*`domain_name, message_ID`*`)`

Parameter

Parameter	Definition
domain_name	file name for message catalog Use only characters that are permitted in file names.
message_ID	text string that identifies a message

Description

The dget_text() function returns a message that has a text string as the message ID. The get_text() and dcget_text() functions serve the same purpose, but dget_text() requires that you specify *domain_name*. You do not specify a locale category for dget_text(), because this function uses the MESSAGES category of the current locale. The *domain_name* and *category* determine the name of the catalog file and the directory path according to the following format:

PATROL_HOME/**lib/nls/locale**/*locale_name*/**ML_***category*/*domain_name***.bo**

The dget_text() function returns *message_ID* when it cannot find a message that matches the relevant *message_ID*, *domain_name* and *category*. It also returns *message_ID* when the current locale is the same as the CTYPE locale.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Example

The following is an example of the dget_text() function.

```
function display_dget_text(domain, msg_id)
{
   ret = dget_text(domain, msg_id);
   printf("Message = %s \n", ret);
 }

display_dget_text("test_msg", "hello");
```

difference()

Return the list of elements that are unique to a specified PSL list

Format

`difference(`*`list1`*`,`*`list2`*`,[`*`list3`*`,`*`list4`*`,...,`*`listn`*`])`

Parameters

Parameter	Definition
list1	a PSL list whose elements are being compared against the elements of all other specified lists
list2...listn	one or optionally more lists whose elements are compared against *list1*

Description

The `difference()` function returns a PSL list with all elements of *list1* that are not in any of the lists *list2*...*listn*. If *list1* is the NULL list, the result is the NULL list.

Duplicates are allowed in *list1*, and they appear in the return list in same order and number as they appeared in *list1*, provided that they were not removed by matches with the other lists in the `difference()` function.

All elements that are returned from *list1* remain in the same order in the return list. If the return list is not the NULL set, the returned set is delimited by new-line characters, that is, all set elements end with a new-line character.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the `difference()` function to find the elements that are unique to *list1*:

```
function main() {
   list1 = [1,2,3,4,5,6,7];
   list2 = [3,4,5];
   list3 = [4,5,6];
   unique1 = difference(list1,list2,list3);
   print("Elements unique to list1 :\n",unique);
}
```

The example produces the following output:

```
Elements unique to list1 :
1
2
7
```

BMC Software, Inc., Confidential and Proprietary Information

dump_hist()

Return the PATROL agent history data for an application, instance, or parameter

Format

```
dump_hist([outfile],[start],[end],[format],
      [application],[instance],[parameter])
```

Parameters

Parameter	Definition
outfile	name and path of the `dump_hist()` function output file If an *outfile* is not specified, `dump_hist()`returns the output as a PSL list. The last member of the list has the following format: `dump_hist: Total matched parameters: n`
start	start of the history interval in the format MMddhhmmyyyy • MM=month • dd=day • hh=hours • mm=minutes • yyyy=year (optional) If the *start* time is not specified, the `dump_hist()` function starts with the oldest available history.
end	end of the history interval in the format MMddhhmmyyyy • MM=month • dd=day • hh=hours • mm=minutes • yyyy=year (optional) If the *end* time is not specified, the `dump_hist()` function ends with the most recent history available.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
format	control characters that specify the content and format of the output for each data point. The *format* parameter permits the following output control options: %H print the agent name %A print the application name %I print the instance name %P print the parameter name %t print the timestamp in elapsed seconds since midnight 01/01/1970 %v print the value of history point %g convert the output time values to Greenwich Mean Time (GMT) %c print the date and time in the format Wed Jul 1 21:45:56 1998 %d print the day of month (1–31) %m print the month of year (1–12) %y print the 4-digit year %h print the hour (0–23) %M print the minutes(0–59) %s print the seconds (0–61) \n add a newline \r add a carriage return \t add a tab \\ print a backslash If the *format* is not specified, the default *format* of "\t%c%v\n" is used with a heading for each parameter containing the agent name, application name, instance name, and parameter name.
application	character string indicating the application If the *application* is not specified, the history for every application is returned.
instance	character string indicating the instance If the *instance* is not specified, the history for each *instance* of the *application* is returned.
parameter	character string indicating the parameter If the *parameter* is not specified, the history for each *parameter* of the *instance* is returned.

Description

The `dump_hist()` function returns the PATROL Agent history data for the specified applications, instances, and parameters. Since it may take a long time to dump the history data, the `dump_hist()` function dumps five parameters and then gives control back to the CPU for other processes, and after 300 milliseconds, the `dump_hist()` function returns to the run queue. This function saves the output to the file specified by the outfile parameter. If outfile is not specified, `dump_hist()` returns the output as a PSL list.

If the `dump_hist()` function cannot open the outfile, the process terminates and returns the following message:

```
Cannot open output file to write
```

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL statement dumps the history for the CACcachCopyReadHitsPercent parameter to **C:\out\dump.txt** using the default output format:

```
outfile="c:\\out\\dump.txt";
start="052000001998";
end="052010001998";
format="";
application="NT_CACHE";
instance="NT_CACHE";
parameter="CACcachCopyReadHitsPercent";
print(dump_hist(outfile,start,end,format,application,instance,parameter));
```

The example produces the following output:

```
pfarrel2/NT_CACHE.NT_CACHE/CACcachCopyReadHitsPercent
        Wed May 20 00:00:27 1998 100
        Wed May 20 00:01:27 1998 100
        Wed May 20 00:02:27 1998 100
        Wed May 20 00:03:27 1998 98.0211
        Wed May 20 00:04:27 1998 100
        Wed May 20 00:05:27 1998 100
        Wed May 20 00:06:27 1998 100
        Wed May 20 00:07:27 1998 100
        Wed May 20 00:08:27 1998 100
```

BMC Software, Inc., Confidential and Proprietary Information

encrypt()

Encrypt a supplied string according to the specified encryption type

Format

`encrypt(`*`str`*`,`*`type`*`)`

Parameters

Parameter	Definition
str	the string you want to encrypt
type	the encryption type of *str* **Valid Values** DES–Returns a DES encryption of the supplied *str* PEM–Returns an encryption of *str* for use in PATROL Event Manager functions **Default** DES If *type* is specified as something other than DES or PEM, DES is also used.

Description

The `encrypt()` function encrypts the supplied string *str* according to the encryption *type* parameter. The *type* parameter can have one of two values, DES or PEM. If *type* is DES, the function returns a DES encryption of the supplied string *str*.

If *type* is PEM, the function returns an encryption of the DES encryption of *str*. Use the PEM encryption *type* when using the string in PATROL Event Manager functions, such as `remote_open()`.

If you are running the stand-alone compiler/interpreter, the `encrypt()` function always returns the empty string ("").

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example shows how to encrypt a string:

```
# Open a connection to a remote Agent
# and prompt for the username and password
user_response = response("Username/Password",-1,"w=300,h=150",
          [R_TEXT_FIELD_LABEL,"Username"],
          [R_TEXT_FIELD_LABEL,"Password"," ","e=0"]);

# Check that the OK button was pressed
status = trim(nthline(user_response, 1),"\n");
if (status) {
   # Extract the username and password
   username = trim(nthline(user_response, 2),"\n");
   password = trim(nthline(user_response, 3),"\n");

   # Encrypt the password
   encrypted_passwd = encrypt(password, "PEM");

   # Open the remote connection
   sess = remote_open(host,port,"PEM",username,encrypted_passwd);
}
```

BMC Software, Inc., Confidential and Proprietary Information

event_archive()

Archive PATROL events that match the specified filter criteria

Format

```
event_archive(filename,operation,[separator],
      [format],[start-time],[stop-time],[status],
      [type],[node],[origin],[pattern],[IDrange],
      [class],[severity],[max-count])
```

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
filename	string that is the name of the file where the archived events are written If the file name begins with a path separator (`/` or `\` depending on the OS), the `event_archive()` function assumes that the path name is a full path name. Otherwise, the `event_archive()` function assumes that the file name is a file in the `$PATROL_HOME/log` directory.
operation	string that specifies the file access used for archiving events **Valid Values** `W` overwrite existing file contents (if any) `A` append to existing file contents (if any)
separator	character string used to separate events in the archive file If `"  "` is specified, the `event_archive()` function uses the newline character (`\n`) to separate events.
format	format string used to present each event entry See “Specifying the event_query() Output Format” on page 1-124. **Default** " " equivalent to "%s %s %s %s %s %s %s %s\n" where the following eight strings are returned in order: • event ID assigned by the PEM • event status • event type • event timestamp • host name that produced the event • application class or instance that produced the event • text string from the event description field • text string from the event diary field

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
FILTER:	
start-time	time endpoint that specifies the oldest event timestamp that is valid for the event archival **Valid Values** " " indicating January 1, 1970 at 00:00:00 or one of the following strings: • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* where the valid ranges of the arguments are: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format, the current year is used when *yy* is omitted.
stop-time	time endpoint that specifies the most recent event timestamp that is valid for the event archival **Valid Values** " " indicating all event timestamps in the repository or one of the following strings: • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* where the valid ranges of the arguments are: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format, the current year is used when *yy* is omitted.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
status	event status that is valid for the event archival **Valid Values** O OPEN A ACKNOWLEDGED C CLOSED E ESCALATED D DELETED For example: • `"O,A,C,D"` matches all statuses except ESCALATED • `"O,A,C,E,D"` or `" "` matches all statuses • `"O,C"` matches only statuses OPEN and CLOSED
type	event type that is valid for the event archival **Valid Values** I INFORMATION S CHANGE_STATUS E ERROR W WARNING A ALARM R RESPONSE For example: • `"S,E,W,A,R"` matches all event types except INFORMATION • `"I,S,E,W,A,R"` or `" "` matches all event types • `"W,A"` matches only event types WARNING and ALARM
node	computer system name that is valid for the event archival **Valid Values** `" "` indicating all computer systems listed in the PEM repository or a host name character string.
origin	application instance or class name that is valid for the event archival **Valid Values** `" "` for all application classes or a character string indicating a specific application instance or class.
pattern	character string within the event description field that is valid for the event archival **Valid Values** `" "` indicating any characters or a character string.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
IDrange	string that defines the range of PATROL event IDs that are valid for the event archival **Valid Values** `x` event ID x `x/y` event IDs between and including x and y `-/y` event IDs less than and including y `x/-` event IDs greater than and including x `" "` (empty string) all events where x and y are integers such that $0 \leq x \leq y \leq 2{,}147{,}483{,}647$
class	event class that is valid for the event archival **Valid Values** `" "` indicating all event classes or a character string specifying a specific event class.
severity	lowest event severity that is valid for the event archival **Valid Values** A string containing an integer between 1 and 5 with 5 being the most severe or " " indicating that all event severities.
max-count	maximum number of events to write to the file If more events are available than the *max-count*, the `event_archive()` function returns the string max count reached instead of the OK string. Default: 100

Note

When the two digit year, *yy*, is specified with the PSL compatible time format for the *start-time* and *stop-time* parameters, the function assumes the year is 19*yy*. You should either omit the year when using this format, or use a different date format if you need to specify the year.

BMC Software, Inc., Confidential and Proprietary Information

Description

The event_archive() function writes PATROL events that match the filter criteria from the PATROL event repository to *filename* using *format* and *separator*. The event_archive() function returns the string OK if successful, or the NULL string if not.

Examples

The following example appends events matching the default event filter to the file **/tmp/myarchive** using a new-line *separator*:

```
event_archive("/tmp/myarchive","A","","");
```

The following example overwrites the existing file **$PATROL_HOME/log/myarchive** with the event description text for events that match the default event filter, separated by new-line characters.

```
event_archive("myarchive","W","\n","%{EV_DESC}");
```

The following example overwrites the existing file **$PATROL_HOME/log/myfile** with events matching the specified filter. The event_archive() function uses the default *separator* and *format*.

```
event_archive(
   "myfile",# archive file $PATROL_HOME/log/myfile
   "A",# append events to existing file
   "",# default separator
   "",# default format
   "",# any start time
   "",# any stop time
   "",# any status
   "",# any type
   "",# any node
   "",# any origin
   "",# any event description
   "",# all event IDs
   "",# any event class
   "2"# event severity 2 or greater
);
```

BMC Software, Inc., Confidential and Proprietary Information

event_catalog_get()

Return information from a PATROL event catalog

Format

`event_catalog_get(`*`catalog`*`,`*`class`*`,`*`format`*`)`

Parameters

Parameter	Definition
catalog	event catalog name whose information is retrieved To use the PATROL STANDARD catalog, specify STANDARD or STD.
class	event class name whose information is retrieved
format	format string used to present event catalog information See “Specifying the event_catalog_get Output Format” on page 1-112.

Description

The event_catalog_get() function returns a text string containing information from a PATROL event catalog. The event_catalog_get() function can return the following information:

- expert advice
- snmp support flag
- event description text
- escalation, notification, and acknowledge command scripts

BMC Software, Inc., Confidential and Proprietary Information

Specifying the event_catalog_get Output Format

The event_catalog_get() function *format* parameter is similar to the specification string for the standard C library printf() function. The *format* parameter can contain alphanumeric characters for use as titles and field names, and string literals for spacing, tabbing, and carriage control.

PATROL macro variables within the *format* parameter identify the fields that the event_catalog_get() function returns. Table 1-6 describes the macro variables available to the event_catalog_get() function.

Note

The event_catalog_get() function macro variables return the same information that appears in the PATROL Developer Console Event Editor dialog box.

Table 1-6 event_catalog_get() Function Macro Variables

Macro Variable	Definition
%{EV_ACK_TEXT}	text string from the acknowledge command of this event catalog and class
%{EV_CTG_DESC}	text string from the description of this event catalog and class
%{EV_ESCL_TEXT}	text string from the escalation command of this event catalog and class
%{EV_EXPERT_ADVICE}	text string from the expert advice of this event catalog and class
%{EV_NOTIFY_TEXT}	text string from the notification command of this event catalog and class
%{EV_SNMP_SUPPORT}	text from the SNMP support of this event catalog and class **Valid Values** NO_TRAP SEND_TRAP

Examples

The following event_catalog_get() function retrieves the text of the Escalation Command edit window for event catalog ORACLEx and event class ResetCounters:

```
event_catalog_get("ORACLEx","ResetCounters","%{EV_ESCL_TEXT}");
```

The following event_catalog_get() function also retrieves the text from the Notification Command edit window and formats each field in order to identify it in the output:

```
event_catalog_get(
   "ORACLEx",
   "ResetCounters",
   "Escalation: %{EV_ESCL_TEXT}\nNotification: %{EV_NOTIFY_TEXT}"
);
```

BMC Software, Inc., Confidential and Proprietary Information

event_check()

Return the number of events in the PATROL Event Manager (PEM) repository that match the specified criteria

Format

```
event_check(start-time,stop-time,status,type,
      node,origin,pattern,IDrange,class,severity)
```

Parameters

Parameter	Definition
start-time	time endpoint that specifies the oldest event timestamp that is valid for the event check **Valid Values** • " " indicating January 1, 1970 at 00:00:00 • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The variables have the following values:: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format the current year is used when *yy* is omitted.
stop-time	time endpoint that specifies the most recent event timestamp that is valid for the event check **Valid Values** • " " indicating January 1, 1970 at 00:00:00 • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format the current year is used when *yy* is omitted.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
status	event status that is valid for the event check **Valid Values** O OPEN A ACKNOWLEDGED C CLOSED E ESCALATED D DELETED **Examples** • `"O,A,C,D"` matches all statuses except ESCALATED • `"O,A,C,E,D"` or `" "` matches all statuses • `"O,C"` matches only statuses OPEN and CLOSED
type	event type that is valid for the event check **Valid Values** I INFORMATION S STATE_CHANGE E ERROR W WARNING A ALARM R RESPONSE **Examples** • `"S,E,W,A,R"` matches all event types except INFORMATION • `"I,S,E,W,A,R"` or `" "` matches all event types • `"W,A"` matches only event types WARNING and ALARM
node	computer system name that is valid for the event check **Valid Values** • `" "` indicating all computer systems listed in the PEM repository • a character string indicating a specific computer system name
origin	application instance or class that is valid for the event check **Valid Values** • `" "` for all application classes • a character string indicating a specific application instance or class

Parameter	Definition
pattern	character string within the event description field that is valid for the event check **Valid Values** • `"  "` indicating any characters • a character string specifying a specific text pattern
IDrange	string that defines the range of PATROL event IDs that are valid for the event query **Valid Values** `x` event ID x `x/y` event IDs between and including x and y `-/y` event IDs less than and including y `x/-` event IDs greater than and including x `-/-` all events where x and y are integers such that $0 \le x \le y \le 2{,}147{,}483{,}647$
class	event class that is valid for the event check **Valid Values** • " " indicating all event classes a character string specifying a specific event class
severity	lowest event severity that is valid for the event archival **Valid Values** • a value between 1 and 5 with 5 being the most severe • " " indicating that all event severities

Note

When the two digit year, *yy*, is specified with the PSL compatible time format for the *start-time* and *stop-time* parameters, the function assumes the year is 19*yy*. You should either omit the year when using this format, or use a different date format if you need to specify the year.

BMC Software, Inc., Confidential and Proprietary Information

Description

The `event_check()` function returns an integer indicating the number of events found in the PATROL Agent PATROL Event Manager repository that matched the search criteria or a `0` indicating no matching events were found.

Example

The following is an example of the event_check() function:

```
print(
   "the number of events matching the filter : " .
   event_check(# the filter definition follows:
   "",#  events after January 1, 1970
   "",#  any stop time
   "O",#  only OPEN status
   "A,W",#only ALARM or WARNING types
   "",#  any computer system name
   "SYBASE",#only events originating from SYBASE
   "",#  match any description text
   "120/-",#only events whose IDs are >= 120
   "" #  any event class
   "2"#  event severity 2 or greater

   )
   . "\n"
);
```

event_query()

Return a list of events in the PATROL Event Manager (PEM) repository that match specified filter criteria

Format

```
event_query(maxcount, delimiter, format,
      [start-time],[stop-time],[status],[type],
      [node],[origin],[pattern],[IDrange],[class],
      [severity])
```

Parameters

Parameter	Definition
maxcount	string that is the maximum number of events that will be returned in the event query Specifying “ “(NULL string) causes *maxcount* to default to 100.
delimiter	string that is used to separate each event in the list of events returned by the event query **Valid Values** • " " indicating that a newline character \n will separate the entries • any valid characters including PSL string literals (see *PATROL Script Language Reference Manual Volume 1—PSL Essentials*)

Parameter	Definition
format	format string used to present each event entry **Default** " " is equivalent to the string "%s %s %s %s %s %s %s %s\n" where the eight strings are returned in the following order: • event ID assigned by the PEM • event status • event type • event timestamp • host name that produced the event • application class or instance that produced the event • text string from the event description field • text string from the event diary field See “Specifying the event_query() Output Format” on page 1-124.
FILTER:	
start-time	time endpoint that specifies the oldest event timestamp that is valid for the event query **Valid Values** • " " indicating January 1, 1970 at 00:00:00 • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format the current year is used when *yy* is omitted.

Parameter	Definition
stop-time	time endpoint that specifies the most recent event timestamp that is valid for the event query **Valid Values** • " " indicating all event timestamps in the repository • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format the current year is used when *yy* is omitted.
status	event status that is valid for the event query. **Valid Values** O OPEN A ACKNOWLEDGED C CLOSED E ESCALATED D DELETED **Examples** • `"O,A,C,D"` matches all statuses except ESCALATED • `"O,A,C,E,D"` or " " matches all statuses • `"O,C"` matches only statuses OPEN and CLOSED

Parameter	Definition
type	event type that is valid for the event query **Valid Values** I INFORMATION S CHANGE_STATUS E ERROR W WARNING A ALARM R RESPONSE **Example** • `"S,E,W,A,R"` matches all event types except INFORMATION • `"I,S,E,W,A,R"` or `"  "` matches all event types • `"W,A"` matches only event types WARNING and ALARM
node	computer system name that is valid for the event query **Valid Values** • `"  "` indicating all computer systems listed in the PEM repository • a host name character string.
origin	application instance or class name that is valid for the event query **Valid Values** • `“"` for all application classes • a character string indicating a specific application instance or class
pattern	character string within the event description field that is valid for the event query **Valid Values** • `"  "` indicating any characters • a character string

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
IDrange	string that defines the range of PATROL event IDs that are valid for the event query **Valid Values** x event ID x `x/y` event IDs between and including x and y `-/y` event IDs less than and including y `x/-` event IDs greater than and including x `-/-` all events x and y are integers in the following ranges: $0 \le x \le y \le 2{,}147{,}483{,}647$
class	event class that is valid for the event query **Valid Values** • `" "` indicating all event classes • character string specifying a specific event class
severity	lowest event severity that is valid for the event archival **Valid Values** • a string containing an integer between 1 and 5 with 5 being the most severe. • " " indicating that events with a severity of 2, 3, 4, and 5 are returned, and events with a severity of 1 are not returned

Note

When the two digit year, *yy*, is specified with the PSL compatible time format for the *start-time* and *stop-time* parameters, the function assumes the year is 19*yy*. You should either omit the year when using this format, or use a different date format if you need to specify the year.

Description

The `event_query()` function returns a list of up to *`maxcount`* events found in the PATROL Agent PATROL Event Manager repository that matched the filter criteria. The returned list is formatted as specified by *`format`*.

BMC Software, Inc., Confidential and Proprietary Information

The `event_query()` function returns the NULL string if no events were found in the event repository that match the filter criteria.

Specifying the event_query() Output Format

The `event_query()` function *`format`* parameter is similar to the specification string used for the standard C library `printf()` function. The *`format`* parameter can contain alphanumeric characters for use as titles and field names, and string literals for spacing, tabbing, and carriage control.

PATROL macro variables within the *`format`* parameter identify the fields that the `event_query()` function returns. Table 1-7 describes the macro variables available to the `event_query()` function.

Table 1-7 event_query() Function Macro Variables (Part 1 of 3)

PEM Macro	Definition
%{EV_ACK_TEXT}	text string from the acknowledge command of this event catalog and class
%{EV_ARG1}	PATROL Event Manager first dynamic argument
%{EV_ARG2}	PATROL Event Manager second dynamic argument
%{EV_ARGS}	character string that presents the event arguments separated by tab characters (`\t`)
%{EV_CATALOG}	name of the PATROL event catalog to which the event belongs
%{EV_CLASS_NAME}	name of the PATROL event class within the PATROL event catalog to which the event belongs
%{EV_CTG_DESC}	text string from the description of this event catalog and class
%{EV_DESC}	the text string description that was produced for the event
%{EV_DIARY}	the text string that was entered into the diary for the event
%{EV_ESCL_TEXT}	text string from the escalation command of this event catalog and class

BMC Software, Inc., Confidential and Proprietary Information

Table 1-7 event_query() Function Macro Variables (Part 2 of 3)

PEM Macro	Definition
%{EV_EXPECTANCY}	life expectancy and disposition of the event. **Valid Values** STORED DEL_IF_CLOSED DEL_IF_INFO DO_NOT_STORE
%{EV_EXPERT_ADVICE}	text string from the expert advice of this event catalog and class
%{EV_HANDLER}	the user ID of the person who performed the last acknowledge, close, or delete action on the event
%{EV_ID}	a sequential integer identifier assigned by the PATROL Event Manager upon receipt of the event
%{EV_NAME}	name of the event within the PATROL event class
%{EV_NODE}	host name that produced the event
%{EV_NOTIFY_TEXT}	text string from the notification command of this event catalog and class
%{EV_NSEVERITY}	numeric severity of the event. Event severity is predefined for all event classes in the STANDARD catalog
%{EV_ORIGIN}	application instance or class that produced the event.
%{EV_OWNER}	user ID that owns the event **Default** PATROL account user ID.
%{EV_SNMP_SUPPORT}	text string from the SNMP support of this event catalog and class **Valid Values** NO_TRAP SEND_TRAP
%{EV_STATUS}	event status **Valid Values** OPEN ACKNOWLEDGED CLOSED ESCALATED DELETED

BMC Software, Inc., Confidential and Proprietary Information

Table 1-7 event_query() Function Macro Variables (Part 3 of 3)

PEM Macro	Definition
%{EV_TIME}	time stamp indicating the system clock time at the moment the event was produced The timestamp is a 24-character string with the following format: `Sun Sep 16 01:03:52 1973`
%{EV_TYPE}	event type **Valid Values** INFORMATION STATE_CHANGE ERROR WARNING ALARM RESPONSE

Example

The following is an example of the `event_query()` function:

```
print("The result of the query" .
   event_query(
      "100",     # return a maximum of 100 events
      "\n\n",     # use two newline characters to separate each event
                 # return ID, type, and description
      "event id: %{EV_ID}\nevent type: %{EV_TYPE}\nevent description:
%{EV_DESC}\n",
                 # filter definition starts here:
      "",        #   any start time
      "",        #   any stop time
      "O",       #   only OPEN status
      "A,W",     #   only ALARM and WARNING types
      "",        #   any node
      "",        #   any origin
      "",        #   match any description text
      "-/-",     #   match any event ID
      "",        #   match any event class
      ""         #   match any event severity
   )
   . "\n"
);
```

BMC Software, Inc., Confidential and Proprietary Information

event_range_manage()

Change the status of PATROL events in a specified event ID range

Format

```
event_range_manage(maxcount,newstatus,username,
      IDrange)
```

Parameters

Parameter	Definition
maxcount	string that is the maximum number of events that will be affected by the status change Specifying the NULL string `"  "` causes *maxcount* to default to 100.
newstatus	string that is the new status to be assigned to each event **Valid Values** ACKNOWLEDGED CLOSED DELETED
username	string that is the user name of the person changing the event status The *username* is placed in each event diary.
IDrange	string that is the range of PATROL event IDs eligible for this `event_range_manage()` function **Valid Values** x event ID x `x/y` event IDs between and including x and y `-/y` event IDs less than and including y `x/-` event IDs greater than and including x `-/-` all events x and y are integers such with the following ranges: $0 \le x \le y \le 2{,}147{,}483{,}647$

BMC Software, Inc., Confidential and Proprietary Information

Description

The `event_range_manage()` function assigns *newstatus* to *maxcount* PATROL events whose event IDs are within *IDrange* and writes *`username`* in the event diary for each of the modified events.

Note

Choose the value of *maxcount* with care. Large *maxcount* values cause more work for the PATROL Agent and can impact its performance.

Managing events using the `event_range_manage()` function follow the same rules used for managing events from the PATROL Console or the PATROL Event Manager (PEM) Console. For example, you cannot acknowledge an event that is closed.

Example

The following event_range_manage() function acknowledges a maximum of 15 events whose event IDs are 30 and above. The user ID used to acknowledge the events is Fred.

```
event_range_manage("15",ACKNOWLEDGED,"Fred","30-");
```

event_range_query()

Return PATROL events from a specified event ID range

Format

`event_range_query(`*`maxcount,delimiter,format,IDrange`*`)`

Parameters

Parameter	Definition
maxcount	string that is the maximum number of events that will be returned in the query Specifying the NULL string `"  "` causes *maxcount* to default to 100.
delimiter	string that is used to separate each event in the list of events returned by the query **Valid Values** • `""` indicating the characters `"		"` will separate the entries • any valid characters including PSL string literals (see *PATROL Script Language Reference Manual Volume 1—PSL Essentials*)

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
format	format string used to present each event returned by the query The format string for the `event_range_query()` function is the same as the format string for the `event_query()` function. See “Specifying the event_query() Output Format” on page 1-124 for details of the format specification. **Default** `" "` is equivalent to the string `"%s %s %s %s %s %s %s  %s\n"` where the eight strings are in the following order: • event ID assigned by the PEM • event status • event type • event timestamp • host name that produced the event • application class or instance that produced the event • text string from the event description field • text string from the event diary field
IDrange	string that is the range of PATROL event IDs eligible for this query **Valid Values** x event ID x `x/y` event IDs between and including x and y `-/y` event IDs less than and including y `x/-` event IDs greater than and including x `-/-` all events where x and y are integers such that $0 \leq x \leq y \leq 2{,}147{,}483{,}647$

Description

The `event_range_query()` function returns up to *maxquery* events from the PATROL Event Manager log whose event IDs are within the range *IDrange*. Each event returned is formatted according to *format*, and the list of events is separated using *delimiter*.

BMC Software, Inc., Confidential and Proprietary Information

The `event_range_query()` function returns the NULL string if no events were found whose event IDs are within *IDrange*.

Note

Choose the value of *maxquery* with care. Large *maxquery* values can impact PATROL Agent performance.

Example

The following is an example of the `event_range_query()` function:

```
event_range_query(
   "100",          # return a maximum of 100 events
   "\n------\n",   # separate events by to newline characters and dashes
                   # return ID, type, status, and description
   "event id: %{EV_ID}\ntype: %{EV_TYPE}\nstatus: %{EV_STATUS}\ndescription:
%{EV_DESC}\n",
   "300/1000"      # only events with IDs such that 300 <= ID <= 1000
);
```

BMC Software, Inc., Confidential and Proprietary Information

event_report()

Return statistics for PATROL events that match the specified filter

Format

```
event_report([start-time],[stop-time],[status],
      [type],[node],[origin],[pattern],[IDrange],
      [class],[severity])
```

Parameters

Parameter	Definition
start-time	time endpoint that specifies the oldest event timestamp that is valid for the event report **Valid Values** • " " indicating January 1, 1970 at 00:00:00 • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format the current year is used when *yy* is omitted.
stop-time	time endpoint that specifies the most recent event timestamp that is valid for the event report **Valid Values** • " " indicating all event timestamps in the repository • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format the current year is used when *yy* is omitted.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
status	event status that is valid for the event report **Valid Values** O OPEN A ACKNOWLEDGED C CLOSED E ESCALATED D DELETED **Examples** • `"O,A,C,D"` matches all statuses except ESCALATED • `"O,A,C,E,D"` or `" "` matches all statuses • `"O,C"` matches only statuses OPEN and CLOSED
type	event type that is valid for the event report **Valid Values** I INFORMATION S CHANGE_STATUS E ERROR W WARNING A ALARM R RESPONSE **Examples** • `"S,E,W,A,R"` matches all event types except INFORMATION • `"I,S,E,W,A,R"` or `" "` matches all event types • `"W,A"` matches only event types WARNING and ALARM
node	computer system name that is valid for the event report **Valid Values** • `" "` indicating all computer systems listed in the PEM repository • a character string indicating a specific host name
origin	application instance or class name that is valid for the event report **Valid Values** • `" "` for all application classes • a character string indicating a specific application instance or class

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
pattern	character string within the event description field that is valid for the event report **Valid Values** • `"  "` indicating any characters • a character string
IDrange	string that defines the range of PATROL event IDs that are valid for the event report **Valid Values** x event ID x `x/y` event IDs between and including x and y `-/y` event IDs less than and including y `x/-` event IDs greater than and including x `-/-` all events x and y are integers in the following range: $0 \le x \le y \le 2{,}147{,}483{,}647$
class	Event class that is valid for the event report **Valid Values** • `"  "` indicating all event classes • a character string specifying a specific event class
severity	lowest event severity that is valid for the event archival **Valid Values** • a string containing an integer between 1 and 5 with 5 being the most severe • " " indicating that all event severities

Note

When the two digit year, *yy*, is specified with the PSL compatible time format for the *start-time* and *stop-time* parameters, the function assumes the year is 19*yy*. You should either omit the year when using this format, or use a different date format if you need to specify the year.

BMC Software, Inc., Confidential and Proprietary Information

Description

The event_report() function returns statistics for events in the PATROL event repository that match the specified filter criteria. The event_report() function returns the empty string if no events match the filter.

Output

The following is a sample of the statistics returned by the event_report() function:

```
Number of events in the Repository: 821
Number of events in cache: 134
Number of events on disk: 687

Status
  OPEN: 821
  ACKNOWLEDGED: 0
  CLOSED: 0
  ESCALATED: 0
  DELETED: 0

Type
  ALARM: 3
  WARNING: 10
  ERROR: 0
  STATE_CHANGE: 806
  INFORMATION: 2
  RESPONSE: 0

Time
  First event: Tue Mar 19 03:34:43 1996
  Last  event: Tue Mar 19 08:25:51 1996

Event Id
  Smallest Id: 14595
  Greatest Id: 16069

Event size:
  Average event size: 225 bytes
  Maximum event size: 287 bytes
```

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the event_report() function:

```
print(
   "event statistics\n",
   event_report(     # filter definition starts here:
      "",            #   any start time
      "0407101596",  #   only events before 04-07-96 10:15
      "O",           #   only OPEN status
      "A",           #   only ALARM type
      "",            #   any computer system
      "SAP",         #   only event originating from SAP
      "",            #   any description text
      "-/-",         #   any event ID
      ""             #   any event class
      ""             #   any event severity
   )
   . "\n"
);
```

BMC Software, Inc., Confidential and Proprietary Information

event_schedule()

Schedule a PATROL event

When you schedule a PATROL event, you are scheduling the event's Notification command (if any) for execution by the PATROL Agent.

Formats

```
event_schedule(add,time,origin,catalog,class,
      type,severity,[argument1],...,[argument20])

event_schedule(delete,scheduleID)

event_schedule(list)

event_schedule(modify,scheduleID,time,origin,
      catalog,class,type,severity,
      [argument1],...,[argument20])

event_schedule(resume,scheduleID,time)

event_schedule(suspend,scheduleID)
```

Parameters

Parameter	Definition
scheduleID	event identifier within the scheduling queue on which the `event_schedule()` function will act. *scheduleID* is returned by the `event_schedule("add")` function

Parameter	Definition
time	string specifying the time the PATROL Agent will schedule the event **Valid Values** RFC-822: *day date month year hh:mm:ss* *timezone* Unix: *day month date hh:mm:ss* *timezone* *year* PSL `date()`: *day month date hh:mm:ss year* where the valid ranges of the arguments are: *day*—Sun Mon Tue Wed Thu Fri Sat *date*—1 to 31 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *hh:mm:ss*—00 to 23, 00 to 59, and 00 to 59 *year*—1902 to 2037 *timezone*—GMT UT EST EDT CST CDT MST MDT PST PDT military time zones A to I, K to Z The *time* parameter must be a minimum of one minute from the time the event is scheduled.
origin	character string containing the application class or instance name that is identified as having created the event
catalog	character string name of the catalog to which the event belongs The `event_schedule()` function requires that *catalog* be previously defined to the PATROL Event Manager. To use the PATROL standard event catalog, specify **STANDARD** or **STD**.
class	character string name of the class to which the event belongs The `event_schedule()` function requires that *class* be previously defined to the PATROL Event Manager.
type	the event type as displayed in the PATROL Event Manager **Valid Values** ALARM WARNING ERROR CHANGE_STATUS INFORMATION

Parameter	Definition
severity	integer value indicating the severity of the event **Valid Values** 1–5 (5 is most severe).
argumentn	arguments passed to the PATROL event description This function permits a maximum of 20 arguments.

Description

The event_schedule() function handles the scheduling of events in the PATROL Agent. Scheduling an event means inserting it into the PATROL Agent schedule queue for triggering at a specified time. When the event is triggered, its notification command is executed. Scheduling an event implies that you are scheduling the OS or PSL Notification command corresponding to that event for execution. The following topics describe each of the event_schedule() function options.

event_schedule() Function Add Option

The event_schedule("add") function causes the PATROL Agent to add the specified event to the PATROL Agent schedule queue for triggering at *time*. The event_schedule("add") function returns *scheduleID* if successful, or the empty string if not.

event_schedule() Function Delete Option

The event_schedule("delete") function removes the event identified by *scheduleID* from the PATROL Agent schedule queue. The *scheduleID* is returned by the event_schedule("add") function. The event_schedule("delete") function returns the string OK if successful, or an error messages string if not.

event_schedule() Function List Option

The event_schedule("list") function returns a list of events contained in the PATROL Agent schedule queue. The following is a sample of the output from the event_schedule("list") function:

```
 ID  TIME                 STATUS
123  Fri Mar 15 15:57:31  active
126  Fri Mar 15 15:57:31  suspended
226  Fri Mar 16 15:57:31  suspended
```

event_schedule() Function Modify Option

The event_schedule("modify") function modifies the properties of an event in the PATROL Agent schedule queue. The *scheduleID* is returned by the event_schedule("add") function. The event_schedule("modify") function returns the string OK if successful, or an error message string if not.

event_schedule() Function Resume Option

The event_schedule("resume") function reinstates an event that has been suspended in the PATROL Agent schedule queue. The *scheduleID* is returned by the event_schedule("add") function. The event_schedule("resume") function returns the string OK if successful, or an error message string if not.

event_schedule() Function Suspend Option

The event_schedule("suspend") function suspends an event in the PATROL Agent schedule queue. The PATROL Agent does not send the suspended event and the event remains in the PATROL Agent schedule queue in the suspended state. The *scheduleID* is returned by the event_schedule("add") function. The event_schedule("suspend") function returns the string OK if successful, or an error message string if not.

BMC Software, Inc., Confidential and Proprietary Information

Examples

Then following examples show different uses for `event_schedule()`.

Schedule Event Notification

The following event_schedule() function adds an event to the PATROL Agent schedule queue:

```
schedule_id = event_schedule(
   "add",                   # schedule a new event
   "Fri Mar 15 15:57:31",   # trigger on 03-15-96 at 3:57 pm
   "MyOrigin",              # origin is MyOrigin
   "STD",                   # catalog is STANDARD
   "41",                    # class is 41
    "INFORMATION",          # type is INFORMATION
   "3",                     # severity is 3
   "myarg"                  # first dynamic argument is myarg
);
```

Delete Scheduled Event

The following event_schedule() function deletes the event schedule_id (created in the previous example) from the PATROL Agent schedule queue:

```
print(
   "deleting event . . .\n   result : " .
   event_schedule("delete",schedule_id) . "\n")
   . "\n"
);
```

Return a List of Scheduled Events

The following event_schedule() function returns a list of the events in the PATROL Agent schedule queue:

```
print(
   "List of Scheduled Events :",
   event_schedule("list")
   . "\n"
);
```

BMC Software, Inc., Confidential and Proprietary Information

Scheduling PSL or OS Activities

As indicated in the comments, this example shows how to schedule PSL or OS activities using the event_schedule() function:

```
# This example shows how you can schedule PSL or OS activities
# using the PSL event_schedule() function. When an event is triggered,
# the agent executes the Notification command defined in the event
# catalog. This notification command can be PSL or OS.
# This is why when you schedule an event, you are basically
# scheduling its notification command to be executed as 'a job'.
#
# In this example I will schedule 6 events: ("STANDARD","EventArchive").
# When this event is executed it dumps all events matching the filter
# into a file

my_date = "Tu Mar 19 14:57:30 1997";     # must be higher than 'date'
my_new_date = "Tu Mar 19 15:55:30 1997"; # Must be higher than my_date

my_event_catalog = "STANDARD";           # put herre your own event catalog
my_event_Class   = "EventArchive;        # put here your own event class to
schedule
my_dump_file     = "/tmp/events.arc";    # put here desired dump destination

# Schedule 3 events ("STANDARD","EventArchive")
# and store their schedule id in sh1, sh2, sh3
#
# This example schedule same job. In practice this is useless

sh1= event_schedule(
   "add",
   my_date,
   "job1",
   my_event_catalog, my_event_class,
   "INFORMATION","3",
   my_dump_file
);
print("add event :  ->".sh1."<-\n\n");

sh2= event_schedule(
   "add",
   my_date,
   "job2",
   my_event_catalog, my_event_class,
   "INFORMATION","3",
   my_dump_file
);
```

BMC Software, Inc., Confidential and Proprietary Information

```
print("add event :  ->".sh2."<-\n\n");

sh3= event_schedule(
   "add",
   my_date,
   "job3",
   my_event_catalog, my_event_class,
   "INFORMATION","3",
   my_dump_file
);
print("add event :  ->".sh3."<-\n\n");

# List all scheduled events
print("list->\n".event_schedule("list")."<-\n\n");
# Schedule 3 more events for execution at my_new_date
# and store their schedule ids in sh4, sh5, sh6

sh4= event_schedule(
   "add",
   my_date,
   "job4",
   my_event_catalog, my_event_class,
   "INFORMATION","3",
   my_dump_file
);
print("add event :  ->".sh4."<-\n\n");

sh5= event_schedule(
   "add",
   my_date,
   "job5",
   my_event_catalog, my_event_class,
   "INFORMATION","3",
   my_dump_file
);
print("add event :  ->".sh5."<-\n\n");

sh6= event_schedule(
   "add",
   my_date,
   "job6",
   my_event_catalog, my_event_class,
   "INFORMATION","3",
   my_dump_file
);
print("add event :  ->".sh6."<-\n\n");

# Delete job6 and list all jobs
```

BMC Software, Inc., Confidential and Proprietary Information

```
print("delete job 6:  ->".event_schedule("delete",sh6)."<-\n\n");
print("list->\n".event_schedule("list")."<-\n\n");

# Suspend job 2 and list all jobs
print("suspend job 2:  ->".event_schedule("suspend",sh2)."<-\n\n");
print("list->\n".event_schedule("list")."<-\n\n");

# Resume job 2 to be executed at my_new_date instead of my_date
# List all jobs
print("resume job 2:  ->".event_schedule("resume",sh2,
my_new_date)."<-\n\n");
print("list->\n".event_schedule("list")."<-\n\n");

# Modify job 4 and list all jobs
print(
   "modify job 4:  ->".
   event_schedule(
      "modify",
      "4",
      my_new_date,
      "new_job4",
      my_event_catalog, my_event_class,
      "INFORMATION","5",
      my_dump_file
   )
   ."<-\n\n"
);
print("list->\n".event_schedule("list")."<-\n\n");
```

BMC Software, Inc., Confidential and Proprietary Information

event_trigger()

Create an event instance for the PATROL Event Manager

Format

```
event_trigger(catalog,class,type,severity,
      [argument1,...,argument20])
```

Parameters

Parameter	Definition
catalog	character string name of the PATROL catalog to which the event belongs To specify the PATROL standard event catalog, use one of following values: • `STANDARD` • `STD` • `0`
class	character string name of the event in the class to which the event belongs
type	the event type as displayed in the PATROL Event Manager **Valid Values** ALARM WARNING ERROR CHANGE_STATUS INFORMATION
severity	integer value indicating the severity of the event **Valid Values** 0–5 (5 is most severe). A severity of 0 will send an event to the PATROL Agent, but the event will not appear in the PATROL Event Manager.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
argumentn	arguments passed to the PATROL Event Manager and displayed in the event detailed report This function permits a maximum of 20 arguments. **Note:** Avoid using the \" characters in this argument. In some cases, using these characters in this argument can cause a command to fail if the command also uses the EV_DESC and EV_ARGS macros.

Description

The `event_trigger()` function creates an event instance that appears in the PATROL Event Manager. The origin of the event instance is the PSL context under which the `event_trigger()` function executes. The `event_trigger()` function assumes that the event catalog and class are already defined.

The `event_trigger()` and `event_trigger2()` functions are identical except how they identify the event origin:

- The `event_trigger()` function always sets the origin to the context in which it executes.
- The `event_trigger2()` function allows you to specify an origin.

The `event_trigger()` function can be used to develop event reporting within PATROL Knowledge Modules.

The return value of this function is the event ID of the event that was created. The event ID appears in the PATROL Event Manager.

Example

The following is an example of the event_trigger() function:

```
event_trigger(
   "STD",          # event catalog : STANDARD
   "41",           # event class: 41
   "INFORMATION",  # event type: INFORMATION
   "3",            # severity: 3
   "myarg"         # one dynamic argument
);
```

When entered at the System Output window, this statement displays the event ID of the event that is created by this function. The event ID appears in the PATROL Event Manager.

```
%PSL print(event_trigger("STD","41","INFORMATION","3", "myarg");
```

BMC Software, Inc., Confidential and Proprietary Information

event_trigger2()

Create an event instance with a specific origin for the PATROL Event Manager

Format

```
event_trigger2(origin,catalog,class,type,severity,
      [argument1,...,argument20])
```

Parameters

Parameter	Definition
origin	character string indicating the application instance or class that originated the event
catalog	character string name of the PATROL catalog to which the event belongs To specify the PATROL standard event catalog, use one of following values: • `STANDARD` • `STD` • `0`
class	character string name of the event in the class to which the event belongs
type	the event type as displayed in the PATROL Event Manager **Valid Values** ALARM WARNING ERROR CHANGE_STATUS INFORMATION

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
severity	integer value indicating the severity of the event **Valid Values** 0–5 (5 is most severe). A severity of 0 will send an event to the PATROL Agent, but the event will not appear in the PATROL Event Manager.
argumentn	arguments passed to the PATROL Event Manager and displayed in the event detailed report This function permits a maximum of 20 arguments. **Note:** Avoid using the \" characters in this argument. In some cases, using these characters in this argument can cause a command to fail if the command also uses the EV_DESC and EV_ARGS macros.

Description

The event_trigger2() function creates an event instance that appears in the PATROL Event Manager. The origin of the event instance is *origin*. The event_trigger2() function assumes that the event catalog and class are already defined.

The event_trigger2() and event_trigger() functions are identical except how they identify the event origin:

- The event_trigger2() function allows you to specify an origin.
- The event_trigger() function always sets the origin to the context in which it executes.

The event_trigger2() function can be used to develop event reporting within PATROL Knowledge Modules.

The return value of this function is the event ID of the event that was created. The event ID appears in the PATROL Event Manager.

Example

The following is an example of the event_trigger2() function:

```
event_trigger2(
   "myorigin",        # origin: myorigin
   "ORACLEx",         # event catalog: ORACLEx
   "ResetCounters",   # event class: ResetCounters
   "ALARM",           # event type: ALARM
   "3",               # severity: 3
   "myarg1",          # dynamic argument 1
   "myarg2"           # dynamic argument 2
);
```

BMC Software, Inc., Confidential and Proprietary Information

execute()

Execute a command or ActiveX script of a specified type

Format

`execute(`*`type`*`,`*`text`*`,[`*`instance`*`],[`*`username`*`],[`*`password`*`])`

Parameters

Parameter	Definition
type	command or script type to execute. The *type* can be a command or ActiveX script type.
	Commands
	For commands, *type* can be one of the following: • `OS` (operating system) • any valid user-defined command type
	Scripts
	For scripts, *type* can be one of the following: • `vbscript` • `jscript` • any valid user-defined *type* that supports the Microsoft ActiveX scripting language The script *type* is prefixed with either % or %@: • % indicates that the *text* parameter provides the text of the script • %@ indicates that the *text* parameter provides the path of a file containing the script The script option is only available with the PATROL Agent for Windows NT 4.0.
text	text or path of the command or script See *PATROL Script Language Reference Manual Volume 1—PSL Essentials* for “PSL Here Documents”.
instance	the application instance against which *text* executes when *type* is a command The *instance* parameter does not apply to the execution of scripts. **Default** The application instance that is the nearest ancestor of command.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
username	user name under which *text* executes when *type* is a command The *username* and *password* together must be a valid account on the PATROL Agent host. The *username* parameter does not apply to the execution of scripts.
password	password under which *text* executes when *type* is a command The *password* and *username* together must be a valid account on the PATROL Agent host. The *password* parameter does not apply to the execution of scripts.

Description

The `execute()` function can either submit a command to a command processor or submit an ActiveX script to the PATROL Scripting Host for execution.

Commands

When the `execute()` function submits a command to command processor, the command executes and returns any output that it produces to `stdout` *or* `stderr`. The status of *command* is saved in the PSL variable `exit_status`.

Sometimes it is desirable to submit long-running commands like daemons to run in the background. For more information on submitting commands to run in the background, see the `popen()` function on page 1-265.

ActiveX Scripts

When the `execute()` function submits an ActiveX script to the PATROL Scripting Host, the PSL process is suspended, and the script is sent to the PATROL Scripting Host. When the script finishes, the PSL process returns to the PSL run queue for execution.

Example

The following examples show the two different ways to use the `execute()` function.

Command Execution

The following example submits an SQL statement to collect data from a table:

```
# SQL data is returned into the buffer "data"
data = execute("SQL", "select * from user_objects");
```

Active Script Execution

The following example submits a Java script from a file:

```
execute("%@jscript","C:\\PATROL\\lib\\java\\java01.txt");
```

BMC Software, Inc., Confidential and Proprietary Information

exists()

Verify the existence of a PSL object

Format

`exists(`*`object`*`,[`*`inherit`*`])`

Parameters

Parameter	Definition
object	the alphanumeric identifier for the object whose existence is being verified The *object* is assigned when the object is created.
inherit	boolean expression controls whether *exists* will search the inheritance hierarchy to verify the existence of *object*: **Valid Values** • TRUE– do not search the inheritance hierarchy • FALSE–search the inheritance hierarchy if *object* is not a reference to an absolute object

Description

The `exists()` function returns TRUE if *object* exists, FALSE otherwise. The `exists()` function is useful in application discovery procedures that determine whether a discovered instance has previously been discovered and instantiated in the object hierarchy.

The exists() function searches for any PSL object that matches *object*, including PSL variables, even if the object was not created with the create() function.

BMC Software, Inc., Confidential and Proprietary Information

Example

This example checks to see if the user name "jsmith" has already been created:

```
name = "jsmith";
if (exists(name)) {
   print(name);
} else {
   print("User name does not exist");
}
```

BMC Software, Inc., Confidential and Proprietary Information

exp()

Return the base of the natural logarithms *e* raised to a power

Format

`exp(`*`exponent`*`)`

Parameter

Parameter	Definition
exponent	numeric value to which the natural base *e* is raised

Description

The `exp()` function returns the value $e^{exponent}$ where *e* is the base of the natural logarithms (*e* = 2.71828 . . .).

Example

The following `exp()` function calls print a number of useful physical and mathematical constants:

```
function main() {
   PI = 3.141593;
   printf("        e = %7.4f\n",exp(1));
   printf("      1/e = %7.4f\n",exp(-1));
   printf("      e^2 = %7.4f\n",exp(2));
   printf(" log10(e) = %7.4f\n",log10(exp(1)));
   printf("     pi^e = %7.4f\n",pow(PI,exp(1)));
   printf("     e^pi = %7.4f\n",pow(exp(1),PI));
   printf("  e^(-pi) = %7.4f\n",pow(exp(1),-PI));
   printf(" e^(pi/2) = %7.4f\n",pow(exp(1),PI / 2));
   printf("e^(-pi/2) = %7.4f\n",pow(exp(1),-PI / 2));
}
```

BMC Software, Inc., Confidential and Proprietary Information

fabs()

Return the absolute value of an argument

Format

`fabs(`*`argument`*`)`

Parameter

Parameter	Definition
argument	floating point value whose absolute value is to be determined

Description

The `fabs()` function returns the absolute value of *argument*; that is,

- *argument* if *argument* ≥ 0
- –*argument* if *argument* < 0

Example

The following `fabs()` function prints the absolute value of -12.3456:

```
print(fabs(-12.3456));
```

file()

Test for file existence and return the last modification time

Format

`file(`*`filename`*`,[`*`enhanced`*`])`

Parameters

Parameter	Definition
filename	name of the file whose last modification date is returned
enhanced	optional flag that requests additional file information in the following form: modtime atime ctime mode size numlinks type • *modtime* is the last modification date expressed as the number of seconds since midnight, January 1, 1970 • *atime* is the last access time expressed as the number of seconds since midnight, January 1, 1970 • *ctime* is the last change of status expressed as the number of seconds since midnight, January 1, 1970 • *mode* is the file permissions expressed as an octal integer • *size* is the length of the file expressed as a number of characters • *numlinks* the number of links to the file within the file system • *type* is a character string indicating the file type: FILE— ordinary user data file DIR— directory SPECIAL— character special file BLOCK— block special file FIFO— pipe or FIFO LINK— symbolic link SOCKET— socket (not available on all platforms) UNKNOWN— unknown file type, possibly a LINK or SOCKET on platforms where the Unix `stat()` function cannot determine the type; that is, where S_ISLINK or S_ISSOCK are undefined Currently, the value of 1 for the *enhanced* parameter is only a place holder and any value is valid, but it is recommended that you use a 1 to specify this parameter to accommodate future enhancements to this function.

Description

The `file()` function returns the last modification time of file *filename* as the number of seconds since midnight, January 1, 1970. If the file does not exist, the `file()` function returns the NULL string. This function is useful for testing the existence of a file.

Note

The `file()` function return values depend on the operating system and in some cases the file system. Some non-Unix platforms may not return

all return values or may return one or more meaningless return values. For a specific platform, the `file()` function will generally return the same information as the C programming language `stat()` function.

The user does not need permission to read the file but does require search permission on each directory in the path name leading to *filename*. If the user does not have such permission, the `file()` function fails and returns the NULL string.

When the *enhanced* parameter is specified, the `file()` function returns a more detailed string of information.

Examples

The following examples highlight the usage of the `file()` function.

Print Last Modification Date of the Unix System Password File

```
print (asctime(file("/etc/passwd"))); # modification time
```

Test for the Existence of a File and Print Enhanced Information

```
file_name="e:\\some_file_name.txt";
if (file(file_name)) {
   print("File ",file_name," exists!");
   print("\nEnhanced Information: \n",file(file_name,1));
} else {
   print("File does not exist.");
}
```

The following is the output from the previous example:

```
File e:\some_file_name.txt exists!
Enhanced Information:
972340188 979711200 962992053 666 2459 1 FILE
```

floor()

Return the largest integer that is not larger than the argument

Format

`floor(`*`argument`*`)`

Parameter

Parameter	Definition
argument	numeric argument whose greatest integer lower bound is to be determined

Description

The `floor()` function returns the largest integer that is not greater than *argument*, that is, the greatest integer lower bound for *argument*.

The `floor()` function and the `ceil()` function together bracket *argument* such that the following are true:

If *argument* is an integer, ceil(*argument*) = *argument* = floor(*argument*).

If argument is not an integer, floor(*argument*) < *argument* < ceil(*argument*) and ceil(*argument*) = floor(*argument*) + 1.

Example

The following `floor()` function call prints the integer 12:

```
print(floor(12.3456));
```

BMC Software, Inc., Confidential and Proprietary Information

fmod()

Return the floating point remainder of a division operation

Format

`fmod(`*`dividend`*`,`*`divisor`*`)`

Parameters

Parameter	Definition
dividend	the floating point value whose remainder will be returned after being divided by *divisor*
divisor	the floating point value that will divide *dividend*

Description

The fmod() function returns the floating point remainder of the division (*dividend*)/(*divisor*).

Example

The following `fmod()` function will return the floating point remainder from the division 10.3 divided by 3.1 (which is 1):

```
print(fmod(10.3,3.1) . "\n");
```

fopen()

Open a PSL channel to a file

Format

`fopen(`*`filename`*`,`*`mode`*`,[shr=upd])`

Parameters

Parameter	Definition
`filename`	name of the file to which the PSL channel should be opened.
`mode`	the file access mode **Valid Values** r open for read w truncate to zero length for write or create file for write a open for append to end of file or create for write rb open binary file for read wb truncate binary file to zero length for write or create binary file for write ab open binary file for append at end of file or create binary file for write r+ open for read and write (update) w+ truncate to zero length for read and write or create for read and write a+ open for read and write at end of file or create file for read and write r+b open binary file for read and write (update) w+b truncate binary file to zero length for read and write or create binary file for read and write a+b open binary file for read and write at end of file or create binary file for read and write

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
`shr=upd`	literal switch used only with PATROL Agent for OpenVMS Use this switch to open a file so that another process can write to it. For more information, refer to documentation for OpenVMS. **Note:** When creating a new file on OpenVMS, the default behavior is in effect; a file is created with a new version number.

Description

The `fopen()` function opens a channel to *filename* that provides the access to *filename* from within a PSL process. The `read()`, `write()`, `get_chan_info()`, `share()`, and `close()` functions apply to channels that have been opened to files.

When supported by the underlying operating system, the `fopen()` function performs security checks to determine whether the user name of the calling process has permission for the request. The valid range of *mode* is the same as for the ANSI C `fopen()` function.

If the `fopen()` function is successful, it returns the PSL channel number to *filename*. A failure to open *filename*, such as an operating system problem or invalid *mode*, sets the PSL `errno` value and causes the `fopen()` function to return the NULL string without attempting to open the file.

The `fopen()` function behaves much like the ANSI C `fopen()` function.

File Permission Changes Using Mode a or w

When the target system is running a variant of Unix, opening *filename* using mode `a` or `w` (append or write) will change *filename* permissions and ownership to be those of the user name of the calling process. The change uses the Unix `chmod` and `chown` commands. The new *filename* permission value is 0644 octal, which is read/write for the process user name and read permission for all other users.

When the target system is non-Unix, the file permission changes depend on whether the underlying file system supports the concept of a file owner. If it does, then file ownership is set to the user name of the calling process and write access is restricted to the owner. If it does not, the file will be accessible by any user.

Support for Binary File Access

The `fopen()` function permits binary modes with a `b` character, though there is currently no way within a PSL process to write any form of binary data other than character strings. The `fopen()` function `b` mode may be useful on some non-Unix platforms to prevent the conversion of CR/LF pairs back and forth to new-line characters, as performed by some standard C libraries for text files.

BMC Software recommends that OS/2 file names be opened in binary mode.

Using the fopen Function with the fseek Function

The `fseek()` function may not be meaningful for text files on some non-Unix platforms, so specifying `+` in *mode* may imply `b` regardless of whether `b` is specified. BMC Software recommends that you use the `+` in *`mode`* for the `fopen()` function whenever you require file positioning using the PSL `fseek()` function.

BMC Software, Inc., Confidential and Proprietary Information

Flush a File after Each PSL File Operation

The PSL functions ensure that a file is flushed after every operation so that the well-known bug of doing a write-then-read or read-then-write without an intervening fseek rewind or fflush does not occur in PSL file operations.

fopen() Function and Named Pipes

Do not use the `fopen()` function to open named pipes (FIFOs). Doing so will cause the PATROL Agent to block indefinitely. Instead, use the following `popen()` function to open a channel for reading to the `read_fifo` shell script:

```
popen("OS","read_fifo fifo");
```

where *fifo* is the name of the named pipe you wish to open, and `read_fifo` is the following shell script:

```
#!/bin/sh
#
# read_fifo <fifo>
#
while [ 1 ]; do
   if [ ! -p $1 ]; then
      cat <<- EOF
      cannot open '$1' -- no such file
      EOF
      exit 1
   cat $1
done
```

You can use the channel number returned by the `popen()` function as an argument in the `read()` or `readln()` functions to retrieve the information the `read_fifo` shell script has read from the named pipe.

Example

The following fopen() function call shows the different modes in which a file can be opened:

```
#   open a file for reading
readchan = fopen("/etc/passwd","r");
#   open a file for writing
writechan = fopen("/tmp/abc123","w");
#   open a file for appending
appendchan = fopen("tmp/newtmp123","a");
```

BMC Software, Inc., Confidential and Proprietary Information

fseek()

Set the file position indicator

Format

`fseek(`*`channel`*`,`*`offset`*`,`*`whence`*`)`

Parameters

Parameter	Definition
channel	the file I/O channel returned when the file was opened by the `fopen()` function
offset	number of bytes to be added to *whence* to obtain the file position
whence	standard point within a file to which the offset is added to obtain the new file position **Valid Values** 0 the beginning of the file 1 the current file position 2 the end of the file

Note

Issuing the `fseek()` function against binary files with *whence* = 2 is not supported on all platforms. See also “Microsoft Windows Has Limited Support for fseek()” on page 1-172.

Description

The `fseek()` function sets the *filename* position indicator to the *whence* position plus *offset* bytes.

When the `fseek()` function encounters invalid parameters (either the *whence*, *offset*, or both), the invalid parameters default to zero, and the `fseek()` function raises a run-time error and completes the file seek operation. Normally, after an invalid parameter defaults to zero, the `fseek()` function completes successfully.

The `fseek()` function returns 0 for success and –1 for failure. For an invalid channel, that is, for a pipe channel instead of a file channel, the `fseek()` function returns –1, raises a run-time error, and sets the PSL `errno` variable.

fseek and Append File Mode

Using the `fseek()` function to change the file position indicator in a file opened in append mode, that is, modes `a`, `ab`, or `a+`, will not prevent writes to the end of the file using the `write()` function.

Microsoft Windows Has Limited Support for fseek()

On Microsoft Windows, you cannot use fseek when *whence* is 1 or with a file that has been opened in text mode. Using fseek under these conditions sets the PSL errno variable to the following value:

Numeric Value	Message
138	E_PSL_RTE_FSEEK_RESTRICTED_FILEOPEN_MODE

PATROL also displays the following error message in the system output window:

```
Text mode fseek is not supported. Refer to the PATROL
Script Language Reference Manual.
```

Example

PSL contains no equivalent to the C `rewind()` function, but the following `fseek()` function example is the equivalent of the C function `rewind(channel)`:

```
fseek(channel,0,0);
```

The following `fseek()` function moves the file position indicator 100 bytes past its current location:

```
fseek(chan,100,1);
```

ftell()

Return the file position indicator

Format

`ftell(`*`channel`*`)`

Parameter

Parameter	Definition
channel	the file I/O channel returned when the file was opened by the `fopen()` function

Description

The `ftell()` function returns the file position indicator as the integer number of bytes from the beginning of the file. For an invalid *channel*, that is, a pipe channel instead of a file channel, the ftell() function returns –1, raises a run-time error, and sets the PSL `errno` variable.

The typical result of both the C and PSL versions of the `ftell()` function is the number of characters written to or read from a file, except on those platforms that perform CR/LF ⇔ new-line conversions on text files. However, the value of the `ftell()` function after executing an `fseek()` function to the end of file is usually the total number of characters in the file.

The following PSL functions change the file position indicator:

- fopen()
- fseek()
- read()
- readln()
- write()

BMC Software, Inc., Confidential and Proprietary Information

The `get_chan_info()` function does not change the file position indicator. The `close()` function makes *channel* invalid.

Example

The following ftell() function call will return the current file position indicator for the channel, *chan*:

```
print(ftell(chan) . "\n");
```

full_discovery()

Verify that the process is currently in a full discovery cycle

Format

```
full_discovery()
```

Parameters

This function has no parameters.

Description

The `full_discovery()` function returns TRUE if the PSL script containing it is an application discovery script and it is currently in a full discovery cycle. Otherwise, the `full_discovery()` function returns FALSE.

A full discovery cycle is done after the PatrolAgent's process cache is refreshed. This flag therefore indicates whether the process cache has been refreshed since the last time the script was executed.

Example

The following example tests whether the PSL script is in a full discovery cycle and exits the script if it is not.

```
# If we are not in a full discovery cycle
# we can exit immediately
if (!full_discovery()) {exit;}
```

BMC Software, Inc., Confidential and Proprietary Information

get()

Return the current value of a variable

Format

`get(`*`variable`*`)`

Parameter

Parameter	Definition
variable	name of the variable whose current value will be returned

Description

The `get()` function returns the current value of *variable*. If *variable* is a relative name and does not exist in the context of the PSL script, the `get()` function successively searches each ancestor's context until *variable* is found or until the search fails in the context of the computer.

Note

When you use the get() function to obtain the hostname, **get(hostname)**, the hostname is returned in a different form depending on the operating system. The hostname value may be returned as all upper-case or all lower-case characters.

Example

The following example returns the current status of RDB database `Dev`.

```
get ("/RDB/Dev/status");
```

BMC Software, Inc., Confidential and Proprietary Information

get_chan_info()

Return status information about a file or process channel

Format

`get_chan_info(`*`channel`*`,[`*`flags`*`])`

Parameters

Parameter	Definition
channel	channel name (shared channels), channel number (local channels), or "" indicating all channels (subject to *flags* control)
flags	optional integer value that controls the format and content of the output **Valid Values** 1 global channels only 2 local channels only 3 both global and local channels **Default** 1 The *flags* parameter determines whether this function returns output in the short or long format. Omit *flags* to return the short format and include *flags* to return the long format. The get_chan_info() function ignores the value of *flags* unless *channel* equals "". For example, if *channel* is 208 and *flags* is 1, the function returns information about the local channel even though the 1 flag requests information about only global channels. The existence of the 1 flag, however, causes get_chan_info() to use the long format for the result.

BMC Software, Inc., Confidential and Proprietary Information

Description

The `get_chan_info()` function returns channel information as a string. The string takes the form of the short or long format based on the existence of the *flags* parameter.

Table 1-8 shows the field names for each format and Table 1-9 shows the valid values for each field name.

Table 1-8 Format of the Result of the get_chan_info() Function

Format	Field Names
short	`status details`
long	`name status details type scope read_pid read_name write_pid write_name`

Table 1-9 Valid Values for the Result of the get_chan_info() function (Part 1 of 2)

Field	Valid Values
name	• *scope*=SHARED—channel name • *scope*=LOCAL—local channel number • scope=" "—all shared and local channels
status	OPEN CLOSED
details	• `fopen()` channel—file name that is opened or NONE if no file name is open • `popen()` channel—process ID of the external operating system process to which the channel is attached or –1 if the process has terminated **Warning:** For fopen(), a space in the name of the file or directory path adds an unwanted space to the text string returned by get_chan_info(). An extra space is a problem because the string is delimited by spaces. You can strip the file or path names from the result with a combination of PSL functions such as length(), index() and substr().
type	• PIPE • FILE
scope	• SHARED • LOCAL
read_pid	• process ID of the PSL process waiting to be read from the channel • –1 if no process is waiting

BMC Software, Inc., Confidential and Proprietary Information

Table 1-9 Valid Values for the Result of the get_chan_info() function (Part 2 of 2)

Field	Valid Values
read_name	• name of the process waiting to be read from the channel • NONE if no process is waiting • UNAVAILABLE if the process name is not available
write_pid	• process ID of the PSL process waiting to be written to the channel • –1 if no process is waiting
write_name	• name of the process waiting to be written to the channel • NONE if no process is waiting • UNAVAILABLE if there is a process but the name is not available

The `get_chan_info()` function returns all the fields for each channel even if they do not apply to the particular channel.

When information is returned for more than one channel, the channel descriptions are formatted as a list with one channel per line separated by new-line characters.

If the *flags* parameter is nonnumeric or less than one, the `get_chan_info()` function produces a run-time warning, and the PSL `errno` variable remains zero.

The `get_chan_info()` function returns the NULL string under the following circumstances:

- no channels exist for the given value of *flags*
- a bad channel number or name is specified for *channel*

  In this case, the `get_chan_info()` function also sets the PSL `errno` variable to `40 (E_PSL_GET_CHAN_INFO_BAD_CHANNEL)`.

Examples

Omitting the *flags* parameter in the following example returns descriptions for all global shared channels, but returns no information for local channels:

```
get_chan_info("");
```

The following example returns descriptions for the local, global, or both local and global channels as controlled by the value of *flags:*

```
get_chan_info("",flags);
```

The following example returns the description of the local or global channel specified by the value of *channel*:

```
get_chan_info("channel");
```

The following example prints both the short and long formats:

```
file_name = "E:\\Program Files\\BMC Software\\Patrol3\\lib\\config.default";
chan1 = fopen(file_name,"r");
print("Short form of output\n".get_chan_info(chan1));
print("\n\nLong form of output\n".get_chan_info(chan1,1));
```

The preceding code creates the following output:

```
Short form of output
OPEN E:\Program Files\BMC Software\Patrol3\lib\config.default

Long form of output
200 OPEN E:\Program Files\BMC Software\Patrol3\lib\config.default FILE LOCAL
   -1 <NONE> -1 <NONE>
```

Note

In the preceding output, the *details* field contains spaces in the path directory name. See *details* in Table 1-9 on page 1-179.

get_ranges()

Return the Border, Alarm1, and Alarm2 range settings for a PATROL parameter

Format

`get_ranges(`*`param`*`,[`*`enhanced`*`])`

Parameter

Parameter	Definition
param	name of the PATROL parameter whose Border, Alarm1, and Alarm2 range settings are returned
enhanced	optional flag that requests additional information When the *enhanced* parameter is set to 1, the `get_ranges()` function returns the state associated with the parameter's border and alarm ranges. Currently, the value of 1 for the *enhanced* parameter is only a place holder and any value is valid, but it is recommended that you use a 1 to specify this parameter to accommodate future enhancements to this function.

Description

The `get_ranges()` function returns the current settings of the Border, Alarm1, and Alarm2 ranges for *param*. The current settings of these alarm ranges are based on one of the following values:

- values set by `set_alarm_ranges() function`
- the default KM alarm ranges

The `get_ranges()` function is used in one of the following ways:

- get_ranges(*param*)—displays the numeric range information. If you do not want the state information, do not specify a second parameter.

BMC Software, Inc., Confidential and Proprietary Information

- get_ranges(*param*, 1)—displays the numeric range information and the state associated with the parameter's border and alarm ranges as one of the following:

  OK
  WARN
  ALARM

The return value of `get_ranges()` consists of three lines, each of which defines the range for Border, Alarm1, and Alarm2, respectively. Each line is a space-separated list of six integers and an optional associated state:

```
active  minimum  maximum  trigger  occurrences  recovery-actions  [state]
```

Field	Definition
active	integer state of the range Active flag This corresponds to the **Active** button on the Border or Alarm dialog. **Valid Values** 0 parameter inactive 1 parameter active
minimum	integer value of the range minimum This value corresponds to the position of the **Minimum** slide bar on the Border or Alarm dialog.
maximum	integer value of the alarm range maximum This value corresponds to the position of the **Maximum** slide bar on the Border or Alarm dialog.
trigger	integer value of the range trigger selection This value corresponds to the **Trigger Alarm** field on the Border or Alarm dialog. **Valid Values** 0 Immediately on alarm 1 After alarm has occurred *n* times 2 After all recovery actions fail

BMC Software, Inc., Confidential and Proprietary Information

Field	Definition
occurrences	integer value of the range *n* occurrences This value corresponds to the position of the ***n* times** slide bar and is only valid when *trigger* = 1 (after alarm has occurred *n* times). **Valid Values** 0 – 99
recovery-actions	integer count of the number of recovery actions defined for the range This value corresponds to the number of entries in the **List of Recovery Actions** for the range.
state	the state associated with the border or alarm range This value is only displayed when the *state* parameter is specified for the `get_ranges()` function. **Valid Values** • OK • WARN • ALARM

Example

The following example uses the get_ranges() function to return the ranges of a parameter (A user-defined function PrintRangeInfo() is called to print the returned alarm range information.):

```
function PrintRangeInfo(name,range) {
   local is_active,minimum,maximum,trigger,occurences, recovery_actions;
   print("Range information for ",name,".\n");
   is_active = ntharg(range,"1");
   minimum = ntharg(range,"2");
   maximum = ntharg(range,"3");
   trigger = ntharg(range,"4");
   occurences = ntharg(range,"5");
   recovery_actions = ntharg(range,"6");

   if (is_active) {
      print("Alarm is active.\n");
   } else {
      print("Alarm is not active.\n");
   }
   print("Lower value is ",minimum,".\n");
   print("Upper value is ",maximum,".\n");
   if (trigger == 0) {
         print("Alarm will trigger immediately.\n");
      } elsif (trigger == 1) {
         print("Alarm will trigger after ",occurences," occurences.\n");
      } elsif (trigger == 2) {
         print("Alarm will trigger after all recovery actions fail.\n");
         }
   print("There are ",recovery_actions," recovery actions.\n");
}

function main() {
   ranges = get_ranges("/MEMORY/MEMORY/MEMPageOut");
   border = nthlinef(ranges,"1");
   alarm1 = nthlinef(ranges,"2");
   alarm2 = nthlinef(ranges,"3");
   PrintRangeInfo("Border Alarm",border);
   PrintRangeInfo("Alarm One", alarm1);
   PrintRangeInfo("Alarm Two", alarm2);
}
```

BMC Software, Inc., Confidential and Proprietary Information

get_text()

Get message text

Format

`get_text(`*`message_ID`*`)`

Parameter

Parameter	Definition
`message_ID`	text string that identifies a message

Description

The get_text() function returns a message that has a text string as the message ID. The dget_text() and dcget_text() functions serve the same purpose, but you cannot use get_text() to specify the domain name or locale category. For locale category, get_text() uses the MESSAGES category as the current locale. For domain, get_text uses the domain of the current locale. You can set this domain by using the text_domain() function. See “text_domain()” on page 1-507. The *domain_name* and *category* determine the name of the catalog file and the directory path according to the following format:

PATROL_HOME/**lib/nls/locale**/*locale_name*/**ML_***category*/*domain_name***.bo**

The get_text() function returns *message_ID* when it cannot find a message that matches the relevant *message_ID*, *domain_name* and *category*. It also returns *message_ID* when the current locale is the same as the CTYPE category.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Example

The following is an example of the get_text() function.

```
function display_get_text(msg)
{
   ret = get_text(msg);
   printf("Message = %s \n", ret);
}

display_get_text("hello");
```

BMC Software, Inc., Confidential and Proprietary Information

get_vars()

Return the list of variables for a PSL object

Format

`get_vars([object],[keyword])`

Parameters

Parameter	Definition
object	optional name of the object whose variables are to be listed **Default** current object.
keyword	optional keyword that specifies the information that is to be returned for *object* **Valid Values** • `nodes`—print nodes only • `subnodes`—return subnodes • `leaves` or `" "`—return leaves only (equivalent to the PATROL Version 3.0 function `get_vars(`*object*`)`) • `all` or `1`—return all children (equivalent to the PATROL Version 3.0 function `get_vars(`*object*`,1)`)

Description

The `get_vars()` function returns a list of the variables of *object* or for the current object if *object* is omitted. The `get_vars()` function returns the NULL string if *object* does not exist.

The list of *object* variables is sorted in ascending alphabetical order.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following get_vars() function call returns a list of variables for a PSL object and all its children:

```
#print the variables for the MEMORY object and all its children
print(get_vars("MEMORY","all"));
```

Sample output for this example might be:

```
.
..
__name__
__self__
__type__
active
environment
instances
lastDiscoveryTime
name
numInstances
osUsername
propagateState
status
username
```

BMC Software, Inc., Confidential and Proprietary Information

getenv()

Return the string value of a PSL environment variable

Format

`getenv(`*`variable`*`)`

Parameter

Parameter	Definition
variable	name of the object whose value is to be returned

Description

The `getenv()` function returns the string value of *variable* in the environment of the PSL script. The variable value can be returned from any of the following places:

- the parameter's defined environment variables
- the application's defined environment variables
- the computer's defined environment
- the environment of the PATROL Agent at the start of its execution

The `getenv()` function searches the environment tables in stated order and returns the value of the first matching variable.The `getenv()` function returns the NULL string if *variable* is not defined and sets the PSL `errno` variable to a nonzero value. If the `getenv()` function is successful, it returns the value of *variable* and sets the PSL `errno` variable to zero. Hence, you can use `errno` to distinguish an undefined variable from one that is set to the NULL string.

Note

The `getenv()` function is not implemented by calling the standard C `getenv()` function.

BMC Software, Inc., Confidential and Proprietary Information

Example

This PSL example presents a function that tests whether an environment variable exists.

```
function is_environment_var_defined(name) {
   getenv(name);        # Throw away return value of getenv
   return (errno == 0); # errno is only zero if it is set
}
```

getpid()

Return the process identifier for the current PSL process

Format

```
getpid()
```

Parameters

This function has no parameters.

Description

The `getpid()` function returns the integer PATROL process identifier of the current PSL process.

Example

The following is an example of the `getpid()` function:

```
printf("Current PID is : %d\n",getpid());
```

BMC Software, Inc., Confidential and Proprietary Information

getpname()

Return the name of a PSL process with a specified process identifier

Format

`getpname([pid])`

Parameters

Parameter	Definition
pid	optional integer process identification number The *pid* value must be an integer.

Description

The `getpname()` function returns the name of the PSL process with a specified *pid*. If you do not specify the *pid* of the process you are looking for, then `getpname()` returns the name of the current process that is calling the `getpname()` function. The empty string, " ", is returned by `getpname()` if there is no process with the specified *pid*.

Example

The following is an example of the `getpname()` function:

```
proc_name = getpname(24);            # Returns the name of PSL process 24
current_proc_name = getpname();      # Returns the name of the current PSL
                                     # process
```

BMC Software, Inc., Confidential and Proprietary Information

grep()

Return the lines from a text block that match a regular expression

Format

`grep(`*`regular-expression`*`,`*`text`*`,[`*`vtcibn`*`])`

Parameters

Parameter	Definition
regular-expression	character sequence that defines the pattern that the grep function searches for in *text* The *regular-expression* conforms to the regular expressions defined in the Unix `ed(1)` command and the Unix `regexp(5)` description. Following is a brief summary of several regular expression characters: ^ beginning of line $ end of line . match any single character * match zero or more repetitions of the preceding [] match any of the characters contained within [^] match any characters except those contained within \< match any characters at the beginning of a word \> match any characters at the end of a word Word boundaries are delimited by any characters other than the following: • A through Z • a through z • 0 through 9 • _
text	text that is searched for matches to *regular-expression* The *text* variable can be a text string enclosed in double quotation marks or one or more PSL commands that produce text as output.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
vtcibn	flag that controls some grep() function options **Valid Values** *v* reverses the output of the `grep()` function, causing the `grep()` function to output all lines in *text* that do not contain a match for *regular-expression*. This flag is similar to the Unix grep -v flag. *t* ignore the meaning of special characters in *regular-expression* *i* ignore case when searching for *regular-expression* matches *b* stop searching for *regular-expression* matches after the first regular-expression match is found *c* return the number of matches of *regular-expression* found in *text* n adds the line number to the front of each line returned by this function

Description

The `grep()` function returns a list of the lines in *text* that match *regular-expression*. The *t* and *v* flags may be used either separately or together and may appear in either order.

Note

The `grep()` function does *not* perform file I/O. You must use the `cat()` function for file I/O. For more information about the `cat()` function, see page 1-38.

Examples

```
# search for "martin" substring in /etc/passwd
all_lines = cat("/etc/passwd"); # fill a buffer with passwd
matching_lines = grep("martin",all_lines);

# search for csh users in /etc/passwd
passwd_file_text = cat("/etc/passwd");
csh_users = grep("/bin/csh",passwd_file_text);
non_csh_users = grep("/bin/csh",passwd_file_text,"v");

# search for users who can’t log in (* in the password entry)
no_login = grep("*",passwd_file_text,"t"); # or...
no_login = grep("\\*",passwd_file_text);   # escape the *

# search for users who can log in
login = grep("*",passwd_file_text,"vt"); # or...
login = grep("\\*",passwd_file_text,"v");

# search for "United" in the Declaration of Independence
all_lines = cat("E:\\Not\\Books\\US_Dec.txt");
matching_lines = grep("United",all_lines,"n");
print(matching_lines );

# output from preceding example
1 The Declaration of Independence of The United States of America
150 We, therefore, the Representatives of the United States of America,
154 solemnly publish and declare, That these United Colonies are,
```

BMC Software, Inc., Confidential and Proprietary Information

history()

Return history information from the PATROL history database

Format

`history([parameter],[format],[number])`

Parameters

Parameter	Definition
parameter	name of the object whose history is returned **Valid Values** • an absolute path (**/APP/INST/PARAM**) • a relative path (**/DIFFERENTINST/PARAM**) • `""` or `"."` for the current parameter **Default** current parameter
format	optional character string inside double quotation marks that specifies the format of each `history()` function entry **Valid Values** n return the number of available data points as the first value in the return list t include the time stamp of each entry in the return list v include the value of each history entry in the return list **Default** ntv
number	optional numeric value that limits the number of entries the history function will return **Default** 50.

BMC Software, Inc., Confidential and Proprietary Information

Description

The `history()` function accesses the parameter history database and returns a list containing the number of data points available followed by a number of entries.

The `history()` function returns the empty string, produces a run-time error, and sets the PSL `errno` variable if a bad format character is provided.

Because of the defaults provided in the `history()` function, the following function specifications are equivalent:

```
history(parameter)
history(parameter,"ntv",50)
```

History Output Format

The `history()` function will return any of the following formats, depending on which *format* flags are set:

- *number_entries*\n if the n flag is set
- *value*\n, *time*\n, or *value time*\n if the v, t, and vt flags are set

The `history()` function separates the values of an entry with spaces and successive entries with new-line characters.

You can use the nthline(*list1*) function to get the number of points from the head of the list and also to extract the entries. Entries can be split if necessary into time and data values using the `ntharg()` function. The entries will be single values if either the `t` or `v` flag is absent.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL statements return history information for the MEMPageOut parameter from the PATROL history database:

```
history_information = history("/MEMORY/MEMORY/MEMPageOut","ntv");
number_of_points = trim( nthlinef(history_information,"1"), "\n");
history_points = nthlinef(history_information,"2-");

printf("Number of points %d\n",number_of_points);
printf("History points:\n%s\n",history_points);
```

BMC Software, Inc., Confidential and Proprietary Information

history_get_retention()

Return the history retention value of a PATROL object

Format

`history_get_retention(object)`

Parameter

Parameter	Definition
object	name of the PATROL object whose history retention value should be returned PATROL object names are expressed as strings in the following form: */application/instance/parameter* In this form, *application*, *instance*, and *parameter* are the application class, instance, and parameter names, respectively.

Description

The `history_get_retention()` function returns the history retention value for *object*. The `history_get_retention()` function returns the NULL string and displays a message in the computer icon console window if an error occurs.

Example

The following history_get_retention() function returns the history retention value for the parameter /USERS/USERS/USRNoUser:

```
print(history_get_retention("USERS/USERS/USRNoUser"),"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

index()

Return the starting position of one string within another

Format

`index(text,string)`

Parameters

Parameter	Definition
text	text to be searched for the occurrence of *string* The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
string	one or more characters enclosed in double quotation marks that are to be located within *text*.

Description

The `index()` function returns the position (indicated as an integer) in *text* at which *string* begins, or 0 if *string* does not occur in *text*. The first position in *text* is position 1.

Example

The following PSL statement uses the index() function to find the position of one string within another:

```
sentence = "The quick red fox jumped over the lazy brown dogs";
animal = "fox";

print(index(sentence,animal));
```

BMC Software, Inc., Confidential and Proprietary Information

int()

Return the largest integer that is not greater than the argument

Format

`int(number)`

Parameter

Parameter	Definition
number	Numeric value or numeric variable

Description

The `int()` function returns the largest integer that is not greater than *number*. It is equivalent to the C library `floor(3)` function.

Example

The following statements use the `int()` function to return only the integer portion of a real number. (Each statement returns the integer 3.)

```
print(int(3.14) . "\n");
print(int(3.48) . "\n");
print(int(3.52) . "\n");
print(int(3.75) . "\n");
print(int(3.99) . "\n");
```

BMC Software, Inc., Confidential and Proprietary Information

internal()

Process a command internal to the PATROL Agent

Format

`internal(`*`command`*`)`

Parameter

Parameter	Definition
command	text string that is the command the PATROL Agent should process

Description

The `internal()` function causes the PATROL Agent to process the string *command* internally and in a platform-specific manner. The `internal()` function returns the *command* output if successful. If unsuccessful or if the command is not supported on the specific platform, the `internal()` function returns the NULL string and sets the PSL variable `errno` = 104 (E_PSL_NOT_SUPPORTED).

The `internal()` function is designed to be used for user and process monitoring and resource inquiries that can be handled inside the PATROL Agent. In these cases, the `internal()` function is much more efficient than invoking a separate command that requires a call to the PSL interpreter or some other command processor.

Note

The `internal()` function is proprietary and FOR BMC SOFTWARE INTERNAL USE ONLY. BMC Software does not support its external use.

BMC Software, Inc., Confidential and Proprietary Information

intersection()

Return a list containing elements that are common to all specified lists

Format

`intersection(`*`list1`*`,`*`list2`*`,[`*`list3`*`,...,`*`listn`*`])`

Parameters

Parameter	Definition
list1...listn	two or more PSL lists that are being evaluated for common elements

Description

The `intersection()` function returns a PSL list containing the elements that appear in all the lists *list1*...*listn*. The returned list is not well-defined and will contain duplicates if duplicates were present in all lists in the same number and order. The elements in the list returned by the intersection() function appear in the same order as they were in *list1*. If any lists are the NULL list, the return value is the NULL list; otherwise all entries in the returned list are terminated by a new-line character.

Example

The following intersection() function call returns the elements that are common to four sets. (The common elements are 7 and 11.)

```
list1 = [1,2,3,7,11];
list2 = [4,5,6,7,8,9,10,11];
list3 = [7,11,12,13,14,15,16,17];
list4 = [7,11];
print("common elements are:\n",intersection(list1,list2,list3,list4));
```

BMC Software, Inc., Confidential and Proprietary Information

in_transition()

Wait for a PSL object to change state

Format

`in_transition(`*`object`*`,`*`timeout`*`)`

Parameters

Parameter	Definition
object	name of the object that is being checked for transition between states
timeout	time in seconds that is allowed for the transition between states

Description

When first called, the `in_transition()` function sets a timer equal to *timeout* seconds on application instance *object*. While the timer is running, subsequent `in_transition()` function calls return FALSE.

After the timer has expired, calls to the `in_transition()` function continue to return FALSE until the next full discovery cycle, at which time the `in_transition()` function will return TRUE and reset the timer. The timer is also reset by calls to the PSL `change_state()` function. (For more information about the `change_state()` function, see page 1-42.)

The `in_transition()` function sets the `errno` variable when there is a problem finding the instance specified by the *object* parameter. The `in_transition()` function sets the errno variable under the following conditions:

BMC Software, Inc., Confidential and Proprietary Information

errno Value	Description
12	E_PSL_CREATE_EMPTY *object* = ""
13	E_PSL_CREATE_EMPTY_SYMBOL_TABLE the application specified by *object* is an application but it does not have a symbol table
14	E_PSL_CREATE_INTERNAL_FAILURE an undetermined internal error
15	E_PSL_CREATE_INACTIVE_APPLICATION the application specified by *object* refers to an inactive application
17	E_PSL_CREATE_UNKNOWN_APPLICATION the application specified by *object* is not an application

If an error occurs while looking up the instance the `in_transition()` function returns the NULL string ("").

Example

The following example uses the in_transition() function to set a 3-minute timer against the ORACLE database instance dev, and when the timer expires, the in_transition() function waits for the next full discovery cycle and changes the state of dev to ALARM:

```
# Some database processes APPEAR to be down.
# Wait to make sure that the database is not
# in the middle of a startup or shutdown.

if (!in_transition ("/ORACLE/dev", 180))
{
  change_state("/ORACLE/dev",ALARM,"Database has crashed.");
}
```

isnumber()

Verify that a string is a valid numeric representation

Format

`isnumber(`*`variable`*`)`

Parameter

Parameter	Definition
`variable`	string that is to be evaluated as meeting the criteria for a numeric expression

Description

The `isnumber()` function returns a Boolean value of 1 if *variable* is a string that is considered valid as a number, or “0” if it is not.

A valid number has only digits, periods, or minus signs for every character in *variable*. White space or any other invalid character anywhere in the string causes the `isnumber()` function to return 0. The `isnumber()` function returns 0 for the NULL string.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the isnumber() function to examine a list and report which of the elements are numbers:

```
# +77 is enclosed in quotes to avoid “+” being interpreted as an operator
list = ["0not",34.2,"16.two",37,94,0.3,-2.8,"+77",nonum];
foreach X (list) {
   if (isnumber(X)) {
      print(X," is a number.\n");
   } else {
      print("Sorry, ",X," is not a number.\n");
   }
}
```

The example produces the following output:

```
Sorry, 0not is not a number.
34.2 is a number.
Sorry, 16.two is not a number.
37 is a number.
94 is a number.
0.3 is a number.
-2.800000 is a number.
+77 is a number.
```

BMC Software, Inc., Confidential and Proprietary Information

is_var()

Verify that a PSL object variable exists

Format

`is_var(`*`object`*`)`

Parameter

Parameter	Definition
object	name of the object that is to be verified as a variable

Description

The `is_var()` function returns TRUE if *object* exists and is a variable. The `is_var()` function returns FALSE for the following conditions:

- *object* does not exist
- *object* exists but is not a variable–it is an application instance or a parameter

The `is_var()` function first checks for the existence of a variable in the context of the argument that is supplied to the function, not the current context. If the variable does not exist in the supplied context, the `is_var()` function searches until the variable is found or the search fails at the computer level.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the is_var() function to determine if the PSL object /MEMORY/MEMORY/MEMPageOut/value is a variable:

```
object = "/MEMORY/MEMORY/MEMPageOut/value"
if (is_var(object)) {
   print(object," is a variable.\n");
}
```

join()

Join the supplied strings together and place a separator between each of the strings

Format

`join(`*`separator`*`,`*`str1`*`,`*`str2`*`,[`*`str3`*`,...,`*`strn`*`])`

Parameters

Parameter	Definition
separator	a specified delimiter used to separate the joined strings
str1...strn	two or more strings (up to 99) that are to be joined

Description

The `join()` function joins the supplied strings together and places the *separator* between each of the strings. Use the `join()` function instead of PSL techniques for common string manipulation. This function takes at least three parameters. The `join()` function is equivalent to the following PSL statement:

```
text = str1 . separator . str2 . separator . str3 ...;
```

Example

The following example joins three strings:

```
filename = join("/","/usr","local","bin");
print(filename);
```

This example produces the following output:

```
/usr/local/bin"
```

BMC Software, Inc., Confidential and Proprietary Information

kill()

Terminate a PSL process

Format

`kill(`*`pid`*`)`

Parameter

Parameter	Definition
pid	integer PATROL process identifier for the PSL process that is to be terminated **Default** the process from which the `kill()` function is called

Description

The `kill()` function terminates the PSL process identified by the process identifier *pid*. If *pid* is omitted, the kill() function terminates the process from which it is called.

The `kill()` function returns 0 if the requested process was killed, or 1 if *pid* did not exist or the process was not killed.

Example

The following is an example of the `kill()` function:

```
kill(4546);
```

BMC Software, Inc., Confidential and Proprietary Information

length()

Return the number of characters in a string

Format

`length(text)`

Parameter

Parameter	Definition
text	text to be counted for character length The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.

Description

The `length()` function returns the length in characters of *text*, including new lines. This function differs from num_bytes(), which returns the length of a string in bytes. Due to international codesets, one or more bytes can form a single character.

Example

The following length() function call returns the number of characters in a string:

```
X = "28 characters in this string";
print(length(X),"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

lines()

Return the number of lines in a string

Format

`lines(text)`

Parameter

Parameter	Definition
text	text to be counted for number of lines (that is, new-line characters) The *text* can be the name of a text file, a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.

Description

The lines() function returns the number of new-line characters in *text*. You can also use the lines() function for returning the length of a list, because the items in a list are delimited by new lines.

Examples

In the following example, the lines() function call returns the number of lines in a character string and in a character list:

```
string = "Jack and Jill\nwent up the hill";
print ("Lines in string: ",lines(string),"\n");
list = [5,"lines","in","this","list"];
print("Lines in list: ",lines(list),"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

lock()

Acquire a PSL process lock

Format

`lock(`*`lockname`*`,[`*`mode`*`],[`*`timeout`*`])`

Parameters

Parameter	Definition
lockname	name of the lock that should be acquired
mode	optional control permitted under the lock **Valid Values** s shared r reader w writer x exclusive **Default** x (If the first letter of *mode* is not `s`, `r`, `w`, or `x`, a run-time error occurs and *mode* defaults to `x` (exclusive).)
timeout	optional integer value that specifies the number of seconds before the lock request expires **Valid Values** • *timeout* > 0 number of seconds the lock request is valid • *timeout* = 0 means nonblocking lock request • timeout < 0 means infinite timeout period **Default** -1 (Infinite *timeout)*

Description

The `lock()` function requests a lock with name *lockname*. The *`mode`* of the request specifies either shared (reader) or exclusive (writer) access under the lock. The optional *timeout* specifies the number of seconds the request is valid.

BMC Software, Inc., Confidential and Proprietary Information

The default behavior of `lock()` function is to request an exclusive lock with an infinite timeout period. The `lock()` function returns 1 for success and 0 for failure.

PATROL Locks and PSL

Lock names are global to the PATROL Agent; thus,

- all PSL processes share the same table of locks
- different PSL processes can share parameter lock names to perform concurrent actions

There is no way to enforce lock-naming scope—BMC Software recommends that lock names in PSL scripts be uniquely encoded using the name of the application. This practice will avoid potential clashes with other PSL scripts.

Shared Lock Requests

Shared lock requests for a lock that is currently in share mode are granted—unless there is a waiting write request. Giving priority to a waiting write request prevents the lock mechanism from starving write processes.

Requests for Locks Already Held

It is possible to request a lock that you already hold, although it is not good style:

- requesting a lock that you already hold is ignored
- requesting a shared lock on a lock you already hold with exclusive access is also ignored

Requesting an upgrade to exclusive access of a lock currently held as shared succeeds and upgrades the lock provided you are the only process that is using the shared lock. If you are not the only process using the lock, the `lock()` function immediately returns 0 in non-blocking mode (regardless of the value of *timeout* because blocking would cause immediate deadlock by waiting for yourself!).

BMC Software, Inc., Confidential and Proprietary Information

Rather than using this upgrade feature, BMC Software recommends that you call the `unlock()` function to release the shared lock before attempting to acquire the new exclusive lock. Lock tracing is possible using the `PslDebug` variable (see *PATROL Script Language Reference Manual Volume 1—PSL Essentials*, “PslDebug—Run-Time Error Checking Variable”). `PslDebug` can be useful in debugging multiprocess lock interactions.

Failure of the lock Function

The `lock()` function can fail if

- a nonblocking request fails
- *timeout* is exceeded before the lock is granted

The `lock()` function can fail for an infinite *`timeout`* if

- a special-case upgrade request is granted
- the system has performed some external deadlock correction

Example

The following example uses the `lock()` function to request a read lock, waiting up to three minutes to acquire the lock:

```
if (lock("PASSWD_FILE_LOCK","w",180)) {
   print("Got the lock\n");
} else {
   print("Did not get it\n");
}
```

log()

Enter and information event to the PATROL Event Manager

Format

`log(`*`message`*`,[`*`severity`*`])`

Parameters

Parameter	Definition
message	text string to be recorded in the PATROL Event Manager Event Diary for an information event
severity	optional text string that specifies the severity of the log message

Description

The `log()` function generates an Information Event and inserts *message* into its Event Diary. The event is recorded in PATROL's Event Log. If *severity* is specified, it is recorded as the severity of the event. The `log()` function always returns the NULL string.

The `log()` function is provided for the purpose of backward compatibility with previous versions of PSL and should not be used to send messages to the Version 3.0 PATROL Event Manager. Use the `event_trigger()` function (see page 1-146) to send messages to the Version 3.0 PATROL Event Manager.

Example

The following is an example of the `log()` function:

```
log("Just Testing . . . ","This is not a real log message");
```

BMC Software, Inc., Confidential and Proprietary Information

loge()

Return the natural logarithm of the argument

Format

`loge(`*`argument`*`)`

Parameter

Parameter	Definition
argument	numeric value whose natural logarithm is to be determined **Valid Values** *argument* > 0.

Description

The `loge()` function returns the logarithm of *argument* with respect to the natural logarithm base $e = 2.71828 \ldots$

The output range for the `loge()` function is $-\infty <$ `loge()` $< \infty$.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following loge() function calls print a number of useful physical and mathematical constants:

```
function main() {
   PI = 3.141593;
   printf("loge(PI) = %7.4f\n",loge(PI));
   printf("loge(10) = %7.4f\n",loge(10));
   printf("loge( 2) = %7.4f\n",loge(2));
   printf("loge( 3) = %7.4f\n",loge(3));
   return;
}
```

The example produces the following output:

```
loge(PI) =  1.1447
loge(10) =  2.3026
loge( 2) =  0.6931
loge( 3) =  1.0986
```

BMC Software, Inc., Confidential and Proprietary Information

log10()

Return the logarithm to base 10 of the argument

Format

`log10(`*`argument`*`)`

Parameter

Parameter	Definition
argument	numeric value whose base 10 logarithm is to be determined **Valid Values** *argument* > 0.

Description

The `log10()` function returns the logarithm of *argument* with respect to base 10.

The output range for the `log10()` function is $-\infty <$ `log10()` $< \infty$.

Example

The following log10() function calls print a number of useful physical and mathematical constants:

```
function main() {
   PI = 3.141593;
   printf("log10(PI) = %7.4f\n",log10(PI));
   printf("log10( e) = %7.4f\n",log10(exp(1)));
   printf("log10( 2) = %7.4f\n",log10(2));
   printf("log10( 3) = %7.4f\n",log10(3));
   return;
}
```

The example produces the following output:

```
log10(PI) =  0.4971
log10( e) =  0.4343
log10( 2) =  0.3010
log10( 3) =  0.4771
```

BMC Software, Inc., Confidential and Proprietary Information

matchline()

Compare text to a regular-expression and save the result to user-defined variables

Format

```
matchline(text,regular_expression,[t],variable_1,
   [variable_2,... variable_9])
```

Parameter

Parameter	Definition
text	text to compare to the regular expression
regular_expression	character sequence that defines the pattern that the matchline() function searches for in *text* The *regular_expression* conforms to the regular expressions defined in the Unix `ed(1)` command and the Unix `regexp(5)` description. The following is a brief summary of several regular expression characters: ^ beginning of line $ end of line . match any single character * match zero or more repetitions of the preceding [] match any of the characters contained within [^] match any characters except those contained within \< match any characters at the beginning of a word \> match any characters at the end of a word Word boundaries are delimited by any characters other than the following: • A through Z • a through z • 0 through 9 • _

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
t	optional flag that causes matchline() to return the number of fields that can be saved to variables If you use this flag, you can compare the number returned by matchline() to the actual number of fields saved to variables.
variable_1 [...variable_9]	one or more user-defined variables You can use a maximum of nine variables. Declare and initialize variables before using them to call matchline().

Description

The matchline() function compares text to a regular-expression and saves the result to one or more user-defined variables.

Typically this function returns the number of fields that it successfully saves to variables. You can use this number to determine how many of the variables, used to call matchline(), contain useful information. With the t flag, matchline() returns the number of fields in *text* that match the regular-expression. You can use this number to determine whether you called matchline() with enough variables to capture all matching fields.

This function returns NULL when it encounters an error, and it assigns the following value to the *errno* variable.

errno Value	Description of Failure
44	E_PSL_GREP_BAD_REGEXP The regular-expression contains one or more errors.

Example

The following example demonstrates the matchline() function:

```
errno=0;
input="abcAAdef";
regrep="\\([a-c]+\\)AA\\([d-f]+\\)";
text_var=" ";
text_var2=" ";

ret= matchline(input,regrep,"",text_var);
print("Matchline with no flag and one variable:\n");
print("Return=".ret." errno=".errno." Text Variable=".text_var."\n");

print("\nMatchline with t flag and one variable:\n");
ret= matchline(input,regrep,"t",text_var);
print("Return=".ret." errno=".errno." Text Variable=".text_var."\n");

print("\nMatchline with t flag and two variables:\n");
ret= matchline(input,regrep,"t",text_var,text_var2);
print("Return=".ret." errno=".errno." Variable=".text_var." Variable
2=".text_var2);
```

The following output results from the preceding example:

```
Matchline with no flag and one variable:
Return=1 errno=0 Text Variable=abc

Matchline with t flag and one variable:
Return=2 errno=0 Text Variable=abc

Matchline with t flag and two variables:
Return=2 errno=0 Variable=abc Variable 2=def
```

BMC Software, Inc., Confidential and Proprietary Information

msg_check()

Validate a message ID and catalog

Format

`msg_check(`*`message_ID`*`)`

Parameter

Parameter	Definition
`message_ID`	integer that identifies a message

Description

The msg_check() function checks whether a message is valid and the message catalog is available. This function returns the following values:

Return Value	Definition
0	catalog is available and contains the message
1	catalog is not available
2	catalog is available but does not contain the message
3	an unknown message service error occurred

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example shows a sample message catalog:

```
VENDOR_ID 1
VENDOR    BMC Software
TOOL_ID   23
TOOL      PATROL Demo
VERSION   1.0
DATE      13-Mar-1997
#-----------------------------------------------------
ID       1       DEMO_HELLO       INFO       CURRENT
MESSAGE "Hello World"
```

The following example of the msg_check() function uses the preceding message catalog:

```
print("msg_check[DEMO_HELLO]=", msg_check(DEMO_HELLO),"\n");
print("msg_check[1142785]=", msg_check(1142785), "\n");
```

The previous commands produce the following output:

```
msg_check[DEMO_HELLO]=0
msg_check[9999999]=2
```

BMC Software, Inc., Confidential and Proprietary Information

msg_get_format()

Return format information for a message

Format

`msg_get_format(`*`message_ID`*`)`

Parameter

Parameter	Definition
`message_ID`	integer that identifies a message

Description

The msg_get_format() function returns the format string retrieved from the message catalog. It returns the following string if it cannot retrieve the catalog or message:

`CATALOG: `*`XXX/XXX`*`, MSGNUM: XXX`

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example shows a sample message catalog:

```
VENDOR_ID 1
VENDOR    BMC Software
TOOL_ID   23
TOOL      PATROL Demo
VERSION   1.0
DATE      13-Mar-1997
#-----------------------------------------------------
ID       2       DEMO_GREET       INFO       CURRENT
MESSAGE "G’Day %s, how’s your %s?\n"
```

The following example of the msg_get_format() function uses the preceding message catalog:

```
print("msg_get_format=", msg_get_format(DEMO_GREET), "\n");
print("msg_get_format=", msg_get_format(999999999), "\n");
```

The previous commands produce the following output:

```
msg_get_format=G’Day %s, how’s your %s?
msg_get_format=Error: Patrol-Message-Unknown: Catalog 3b9/ac
   MsgNum 2559
```

BMC Software, Inc., Confidential and Proprietary Information

msg_get_severity()

Get severity information for a message

Format

`msg_get_severity(`*`message_ID`*`)`

Parameter

Parameter	Definition
`message_ID`	integer that identifies a message whose severity you want to retrieve

Description

The msg_get_severity() function returns an integer that indicates the severity of a message. This function gets the severity from the message catalog, which stores a severity for each message. The following list shows return values for msg_get_severity().

Return Value	Description
1	INFO
2	WARNING
3	ERROR
4	FATAL
-1	could not retrieve severity

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example shows a sample message catalog:

```
VENDOR_ID 1
VENDOR    BMC Software
TOOL_ID   23
TOOL      PATROL Demo
VERSION   1.0
DATE      13-Mar-1997
#-----------------------------------------------------
ID       2       DEMO_GREET       INFO       CURRENT
MESSAGE "G’Day %s, how’s your %s?\n"
```

The following example of the msg_get_severity() function uses the preceding message catalog:

```
print("Severity=", msg_get_severity(DEMO_GREET));
```

The previous command produce the following output:

```
Severity=1
```

msg_printf()

Display message text in a computer output window on the PATROL Console

Format

```
msg_printf(message_ID, [argument_1,
   argument_2...argument_N])
```

Parameter

Parameter	Definition
message_ID	integer that identifies a message that you want to print
argument_1... argument_N	values that replace variables or control characters in the text of the message

Description

The msg_printf() function displays the text of a message to the computer or task output window. This function is very similar to printf(), but msg_printf() displays information from the message catalog while printf() displays information that you supply.

The parameters that you use with msg_printf() supply information that the message catalog cannot provide. The *message_ID* identifies which message you want to display. The optional arguments provide values for variables or control characters that exist in the message text. For example, the %d character sequence represents a signed decimal. If a message contains %d in its text, you must supply a signed decimal value as an argument to msg_printf(). When this function displays the message, it displays the decimal (provided by you) in place of the %d. The variables and control characters in these messages follow the same rules as the *format* parameter of the printf() function. See “printf()” on page 1-275.

BMC Software, Inc., Confidential and Proprietary Information

Typically, msg_printf() returns the empty string, but it returns the following error message when it cannot find a message ID or message catalog:

```
CATALOG: XXX/XXX, MSGNUM: XXX
```

When you call the msg_printf() function, a PATROL Console, not a PATROL Agent, retrieves the message and substitutes any arguments. For this reason, the message catalog must be available to the console side of the connection.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Example

The following example shows the msg_printf() function:

```
msg_printf(BMC_KM_ORACLE_ALL_OK, threads_num,
   host_name);
```

The BMC_KM_ORACLE_ALL_OK argument is the ID for the following message:

```
Everything is OK. Total %d threads on host %s.
```

The msg_printf() function replaces the %d and %s characters with the values from the *threads_num* and *host_name* variables. It then displays the preceding message in the computer or task output window.

BMC Software, Inc., Confidential and Proprietary Information

msg_sprintf()

Return message text from a message catalog

Format

```
msg_sprintf(message_ID, [argument_1,
   argument_2...argument_N])
```

Parameter

Parameter	Definition
message_ID	integer that identifies a message that you want this function to return
argument_1... argument_N	values that replace variables or control characters in the text of the message

Description

The msg_sprintf() function returns the text of a message. This function is very similar to sprintf(), but msg_sprintf() returns information from the message catalog while sprintf() returns information that you supply.

The parameters that you use with msg_sprintf() supply information that the message catalog cannot provide. The *message_ID* identifies which message you want to return. The optional arguments provide values for variables or control characters that exist in the message text. For example, the %d character sequence represents a signed decimal. If a message contains %d in its text, you must supply a signed decimal value as an argument to msg_sprintf(). When this function returns the message, the decimal (provided by you) replaces the %d. The variables and control characters in these messages follow the same rules as the *format* parameter of the sprintf() function. See “sprintf()” on page 1-487.

When msg_sprintf() cannot find a message ID or message catalog, it returns the following error message:

BMC Software, Inc., Confidential and Proprietary Information

`CATALOG: XXX/XXX, MSGNUM: XXX`

The operations of the msg_sprintf() function can occur on a PATROL Console or PATROL Agent. Calling msg_sprintf() from a menu command on the console, for example, would run this function on the console. Calling it from a PSL script in the agent, on the other hand, would run this function on the agent. For this reason, you have to make the message catalog available to the component that runs the msg_sprintf() function.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Example

The following example shows the msg_sprintf() function:

```
variable_1=msg_sprintf(BMC_KM_ORACLE_ALL_OK,
   threads_num,host_name);
```

The BMC_KM_ORACLE_ALL_OK argument is the ID for the following message:

```
Everything is OK. Total %d threads on host %s.
```

The msg_sprintf() function replaces the %d and %s characters with the values from the *threads_num* and *host_name* variables. It then returns the preceding message and stores it in *variable_1*.

ntharg()

Return a formatted list containing fields from a text string

Format

```
ntharg(text,arguments,[delimiters],[separator]
   [line_separator])
```

Parameters

Parameter	Definition
text	text to be separated into fields by the `ntharg()` function The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
arguments	integer list specifying the field numbers `ntharg()` should look for in each line of *text*. **Valid Values** x,y field x and field y x-y all fields from x to y inclusive -x all fields from 1 to x inclusive x- all fields from x to the new-line character inclusive
delimiters	one or more optional characters that `ntharg()` should treat as field separators when examining *text* **Default** space and `\t` (tab).
separator	optional character(s) that `ntharg()` inserts between each field of output **Default** new-line character (\n).

Parameter	Definition
line_separator	optional character(s) that ntharg() uses to replace each new-line character (\n) that appears in *text* **Default** new-line character (\n). **Example** If *separator* is "-" and *line_separator* is "?", ntharg() converts "red green\nblue orange\nyellow purple" to "red-green?blue-orange?yellow-purple".

Description

The `ntharg()` function returns the *arguments* in *text* and normally interprets each line in *text* as a white space-separated (space or tab) list of fields. If *delimiters* is given, it specifies the list of characters that `ntharg()` should treat as field separators. The `ntharg()` function normally returns selected fields as a new-line delimited list. If *separator* is given, it specifies the delimiter to be placed between items in the returned list.

Note

The difference between the `ntharg()` function and the `nthargf()` function is as follows:

- The `ntharg()` function treats each delimiter that follows a non-delimiter character as the end of a field. The `ntharg()` function interprets two or more adjacent delimiters as a single delimiter, thereby ignoring multiple delimiters and treating them as one.

- The `nthargf()` function treats each delimiter as the end of a field. The `nthargf()` function interprets two or more adjacent delimiters as delimiting one or more NULL strings whose content can be requested and returned.

BMC Software, Inc., Confidential and Proprietary Information

Examples

The following example prints the login name and home directory of each user listed in the Unix system password file:

```
foreach user (cat("/etc/passwd")) {
   print(ntharg(user,"1,6",":","\t","\n"),"\n");
}
```

The following example illustrates the difference between the ntharg() function and the nthargf() function:

```
string = "abc::123:xyz";
A = ntharg(string,"3",":");
B = nthargf(string,"3",":");
print("ntharg() 3rd string argument is ",A,"\n");
print("nthargf() 3rd string argument is ",B,"\n");
```

The example produces the following output:

```
ntharg() 3rd string argument is xyz
nthargf() 3rd string argument is 123
```

BMC Software, Inc., Confidential and Proprietary Information

nthargf()

Return a formatted string containing fields from a text string

Format

```
nthargf(text,arguments,[delimiters],[separator],
   [line_separator])
```

Parameters

Parameter	Definition
text	text to be separated into fields by the `nthargf()` function The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
arguments	integer list specifying the field numbers `nthargf()` should look for in each line of *text* **Valid Values** x,y field x and field y x-y all fields from x to y inclusive -x all fields from 1 to x inclusive x- all fields from x to the new-line character inclusive
delimiters	one or more characters that `nthargf()` should treat as field separators when examining *text* The `nthargf()` function treats each occurrence of *delimiters* as delimiting a field. The `nthargf()` function interprets two or more adjacent *delimiters* as delimiting one or more NULL fields. **Default** space and `\t` (tab).

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
separator	optional character(s) that should be placed between each field of `nthargf()` output **Default** new-line character (\n).
line_separator	optional character(s) that nthargf() uses to replace each new-line character (\n) that appears in *text* **Default** new-line character (\n). **Example** If *separator* is "-" and *line_separator* is "?", ntharg() converts "red green\nblue orange\nyellow purple" to "red-green?blue-orange?yellow-purple".

Description

The `nthargf()` function returns the *arguments* in *text*. The `nthargf()` function normally interprets each line in *text* as a white space-separated (space or tab) list of fields. If *delimiters* is given, it specifies the list of characters that `nthargf()` should treat as field separators. The `nthargf()` function normally returns selected fields as a new-line delimited list. If *separator* is given, it specifies the delimiter to be placed between items in the returned list.

The difference between the `nthargf()` function and the `ntharg()` function is as follows:

- The `nthargf()` function treats each delimiter as the end of a field. The `nthargf()` function interprets two or more adjacent delimiters as delimiting one or more NULL strings whose content can be requested and returned.

- The `ntharg()` function treats each delimiter that follows a non-delimiter character as the end of a field. The `ntharg()` function interprets two or more adjacent delimiters as a single delimiter, thereby ignoring multiple delimiters and treating them as one.

Example

The following example illustrates the difference between the ntharg() function and the nthargf() function:

```
string = "abc::123:xyz";
A = ntharg(string,"3",":");
B = nthargf(string,"3",":");
print("ntharg() 3rd string argument is ",A,"\n");
print("nthargf() 3rd string argument is ",B,"\n");
```

The example produces the following output:

```
ntharg() 3rd string argument is xyz
nthargf() 3rd string argument is 123
```

BMC Software, Inc., Confidential and Proprietary Information

nthline()

Return specified lines from a text string

Format

`nthline(`*`text`*`,`*`lines`*`,[`*`separator`*`])`

Parameters

Parameter	Definition
text	text to be separated into lines by the `nthline()` function The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
lines	integer list specifying the line numbers `nthline()` should look for in *text* **Valid Values** x,y line x and line y x-y all lines from x to y inclusive -x all lines from 1 to x inclusive x- all lines from x to EOF character inclusive
separator	optional character that should be placed between each field of `nthline()` output **Default** new-line character (\n).

Description

The `nthline()` function returns the *lines* of *text* separated by new-line characters. If you specify a *separator*, the `nthline()` function will use *separator* to separate lines.

Note

The difference between the `nthlinef()` and `nthline()` functions is as follows:

- The `nthlinef()` function treats each new-line character as a line.
- The `nthline()` function treats only a nonempty line (that is, a line with a nonnew-line character preceding a new-line character) as a line.

BMC Software recommends you use `nthlinef()` function to be consistent with other PSL functions.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL script prints the top five processes executing on a Unix system.

```
# print the top five processes
print(nthline(system("ps -eaf"),"2-6"));
```

The following example illustrates the difference between the nthline() function and the nthlinef() function:

```
string = "abc\n\n123\nxyz";
A = nthline(string,3);
B = nthlinef(string,3);
print("nthline() 3rd string argument is  : ",A,"\n");
print("nthlinef() 3rd string argument is : ",B,"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

nthlinef()

Return specified lines from a text string

Format

`nthlinef(`*`text`*`,`*`lines`*`,[`*`separator`*`])`

Parameters

Parameter	Definition
text	text to be separated into lines by the `nthlinef()` function The *text* can be a text string enclosed in double quotes, or one or more PSL commands that produce text as output.
lines	integer list specifying the line numbers `nthlinef()` should look for in *text* **Valid Values** x,y line x and line y x-y all lines from x to y inclusive -x all lines from 1 to x inclusive x- all lines from x to EOF character inclusive
separator	optional character that should be placed between each field of `nthlinef()` output **Default** new-line character (\n).

BMC Software, Inc., Confidential and Proprietary Information

Description

The `nthlinef()` function returns the *lines* of *text* separated by new-line characters. If you specify a *separator*, the `nthlinef()` function will use *separator* to separate lines.

Note

The difference between the `nthlinef()` and `nthline()` functions is as follows:

- The `nthlinef()` function treats each new-line character as a line.
- The `nthline()` function treats only a nonempty line (that is, a line with a nonnew-line character preceding a new-line character) as a line.

BMC Software recommends you use `nthlinef()` function to be consistent with other PSL functions.

Example

The following PSL script prints the top five processes executing on a Unix system:

```
# print the top five processes
print(nthlinef(system("ps -eaf"),"2-6"));
```

The following example illustrates the difference between the nthlinef() function and the nthline() function:

```
string = "abc\n\n123\nxyz";
A = nthline(string,3);
B = nthlinef(string,3);
print("nthline() 3rd string argument is  : ",A,"\n");
print("nthlinef() 3rd string argument is : ",B,"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

num_bytes()

Return the length of a string in number of bytes

Format

```
num_bytes(string, [num_chars])
```

Parameters

Parameter	Definition
string	text string whose final character is NULL
num_chars	number of characters that you want to count

Description

The num_bytes() function returns the number of bytes in a string. It differs from length(), which returns the number of characters. Due to international codesets, a single character can be composed of one or more bytes.

This function counts all bytes in a string until it reaches the NULL character "", which causes the function to return the number of bytes it has counted. You can use the optional *num_chars* parameter to limit the number of characters that the function counts. For example, if *num_chars* is 20, num_bytes() returns 20 if the string is 20 characters or longer. It returns a number less than 20, however, if num_bytes() finds a NULL character before it counts the twentieth character.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

The example on page 1-250 shows that num_bytes() reads characters but returns bytes. The *num_chars* parameter has a value of 4, but num_bytes() returns 6. This difference exists because two of the characters are composed of two bytes.

Example

This example shows the num_bytes() function.

```
# LANG = ja_JP.eucJP
function run_num_bytes(id, str, nchar){
   ret = num_bytes(str, nchar);
   printf("ID = %d, nchar = %d, return = %d\n", id,
nchar, ret);
 }

run_num_bytes("1", "", -1);
run_num_bytes("2", "", 0);
run_num_bytes("3", "", 1);
run_num_bytes("4", "a ", -1);
run_num_bytes("5", "a ", 0);
run_num_bytes("6", "a ", 1);
run_num_bytes("7", "a ", 2);
run_num_bytes("8", "¤Ò¤ç¤≠ ", -1);
run_num_bytes("9", "¤Ò¤ç¤≠ ", 0);
run_num_bytes("10", "¤Ò¤ç¤≠ ", 1);
run_num_bytes("11", "¤Ò¤ç¤≠ ", 2);
run_num_bytes("12", "¤Ò¤ç¤≠ ", 3);
run_num_bytes("13", "¤Ò¤ç¤≠ ", 4);
run_num_bytes("14", "¤Ò¤ç¤≠ ", 10);
```

The preceding example produces the following output:

```
ID = 1, nchar = -1, ret = 0
ID = 2, nchar = 0, ret = 0
ID = 3, nchar = 1, ret = 0
ID = 4, nchar = -1, ret = 1
ID = 5, nchar = 0, ret = 0
ID = 6, nchar = 1, ret = 1
ID = 7, nchar = 2, ret = 1
ID = 8, nchar = -1, ret = 6
ID = 9, nchar = 0, ret = 0
ID = 10, nchar = 1, ret = 2
ID = 11, nchar = 2, ret = 4
ID = 11, nchar = 3, ret = 6
ID = 11, nchar = 4, ret = 6
ID = 11, nchar = 10, ret = 6
```

BMC Software, Inc., Confidential and Proprietary Information

num_consoles()

Return the number of consoles registered to the PATROL Agent or that use the PSL script

Format

```
num_consoles([ALL])
```

Parameter

Parameter	Definition
ALL	optional flag that returns the number of consoles registered with the PATROL Agent. If the ALL flag is omitted, the `num_consoles()` function returns the number of PATROL Consoles that use the PSL script.

Description

The `num_consoles()` function returns a integer representing the number of PATROL Consoles that:

- are registered with the PATROL Agent (`ALL` argument)
- use the script in which the `num_consoles()` function appears (no argument)

The value that the `num_consoles()` function returns when `ALL` is omitted can be any positive integer from 0 to the number obtained from the `num_consoles(ALL)` function, that is, the total number of consoles registered with the PATROL Agent.

The `num_consoles()` function (no `ALL` argument) always returns 1 when the PSL script is used as a menu command.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the num_consoles() function to return the number of consoles connected to the PATROL Agent, and the number of consoles connected to the PATROL Agent that are running the example script:

```
connect = num_consoles(ALL);
scriptrunner = num_consoles();

print("Number of Consoles connected to this Agent : ",connect,"\n");
print("Number of Consoles executing this script   : ",scriptrunner,"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

pconfig()

Manage the PATROL Agent configuration variables

Format

```
pconfig(APPEND,variable,[value])
pconfig(DELETE,variable,[value])
pconfig(FORMAT,[options],[name_separator],
   [value_separator],[variable])
pconfig(GET,variable)
pconfig(LIST)
pconfig(MERGE,variable,[value])
pconfig(PURGE)
pconfig(RELOAD)
pconfig(REPLACE,variable,[value])
```

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
variable	string identifying the PATROL Agent configuration variable on which `pconfig()` is to act The *variable* parameter is validated using the following rules: • *variable* is composed of name segments separated by a slash (/) • *variable* may start with a slash (/), but not end with a slash(/) • *variable* may not contain empty segments • *variable* may not contain unprintable characters • *variable* may not contain white space other than the space character • *variable* may contain spaces within a segment but not at the beginning or end of a segment
value	optional string identifying the value associated with *variable* **Default** the NULL string You can assign multiple values using a comma to separate values in the string as follows: "*value1*,*value2*, . . . , *valuen*"
options	optional bitmask of preferences that specifies the variables to be returned and how they are to be formatted
name_separator	optional string to output after the name of *variable*
value_separator	optional string to output after the value of *variable*

Description

The `pconfig()` function can be used to view and change the configuration variables for a PATROL Agent. Changes made with the `pconfig()` function take effect immediately without restarting the PATROL Agent, and these changes are persistent and will survive a PATROL Agent restart.

BMC Software, Inc., Confidential and Proprietary Information

The following topics describe each of the `pconfig()` options. (See the *PATROL Agent Reference Manual* for more information about PATROL Agent configuration.)

APPEND Option

The `pconfig("APPEND")` function appends *value* to the list of values for *variable*. If *variable* does not exist, the `pconfig("APPEND")` function creates *variable* and adds *value* to it.

The `pconfig("APPEND")` function does not check to see if *variable* already has *value* in its list of values. (If so, the `pconfig("APPEND")` function adds it a second time.)

The `pconfig("APPEND")` function returns the NULL string and sets the PSL variable `errno` = 0 (E_PSL_NO_ERROR) if successful, or a PSL error message string and nonzero `errno` return value otherwise. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

DELETE Option

The `pconfig("DELETE")` option deletes variable information from the PATROL Agent configuration:

- If both *variable* and *value* are specified, the `pconfig("DELETE")` function deletes *value* from *variable*
- If only *variable* is specified, the `pconfig("DELETE")` function deletes *variable* from the PATROL Agent configuration.

If *variable* does not exist in the PATROL Agent configuration, the `pconfig("DELETE")` function performs no operation.

The `pconfig("DELETE")` function returns the NULL string and sets the PSL variable `errno = 0` (E_PSL_NO_ERROR) if successful, or a PSL error message string and nonzero `errno` return value otherwise. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

BMC Software, Inc., Confidential and Proprietary Information

FORMAT Option

The `pconfig("FORMAT")` returns a list of PATROL Agent configuration variables or a single configuration variable in the format that you specify. The `pconfig("FORMAT")` call is similar to `pconfig("GET")` and `pconfig("LIST")` calls, but it can be a little more flexible in some situations.

The *options* parameter is a bitmask of preferences that specifies the variables to be returned and how they are to be formatted. The available option preferences areas follows:

64 Place quotes around name and values
32 Search for given *variable* only
16 Output default variables
8 Output overridden/non-default variables
4 Output variable names
2 Output current values
1 Output default values

Combinations of these preferences can be specified by ORing the values together. For example, if *options* is set to 18 (that is, 16 | 2), then the `pconfig("FORMAT")` option returns the current values of the default variables.

The *name_separator* and *value_separator* parameters specify the strings that should appear after each variable name and each value respectively.

The last parameter *variable* specifies the name of the variable that should be returned. Without *variable*, `pconfig("FORMAT")` is a generalized form of `pconfig("LIST")` but with a specified *variable*.

The following table lists the parameters that are optional along with their default values:

Parameter	Default Value
options	4
name_separator	\n

Parameter	Default Value
value_separator	\n
variable	""

If the `pconfig("FORMAT")` function cannot obtain the list of variables or the value of a variable, it returns a PSL error message string and a nonzero errno return value. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

Examples of the pconfig("FORMAT") Option

```
# Get just the current value
pconfig("FORMAT",32|2,"","",
    "/AgentSetup/historyRetentionPeriod");


# List out only non-default/overriden variables
# with name=value<newline> format
print(pconfig("FORMAT",8|4|2,"=","\n",""));
```

GET Option

The `pconfig("GET")` function returns the default and current values of *variable* in a PSL list with the following format:

default_value\n
current_value

If the specified variable does not exist, the `pconfig("GET")` function returns a NULL string and sets the `errno` variable to a non-zero value.

LIST Option

The `pconfig("LIST")` function returns a list of all the PATROL Agent configuration variables in a PSL list with the following format:

variable1\n
variable2\n
. . .
variablen

If the `pconfig("LIST")` function cannot obtain the list of variables, it returns a PSL error message string and a nonzero `errno` return value.

MERGE Option

The `pconfig("MERGE")` function merges *value* into the list of values for *variable*. If *variable* does not exist, the `pconfig("MERGE")` function creates *variable* and adds *value* to it.

The `pconfig("MERGE")` function checks to see if *variable* already has *value* in its list of values and, if so, performs no action.

The `pconfig("MERGE")` function returns the NULL string and sets the PSL variable `errno` = 0 (E_PSL_NO_ERROR) if successful or a PSL error message string and nonzero `errno` return value otherwise. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

PURGE Option

The `pconfig("PURGE")` function removes the existing configuration from the PATROL Agent and creates a new one based on the PATROL Agent `config.default` file.

The `pconfig("PURGE")` function returns the NULL string and sets the PSL variable `errno` = 0 (E_PSL_NO_ERROR) if successful or a PSL error message string and nonzero `errno` return value otherwise. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

RELOAD Option

The `pconfig("RELOAD")` function reloads the default configuration information from the PATROL Agent `config.default` file only if the timestamp on `config.default` has been updated since the last time it was read by the PATROL Agent. (The PATROL Agent keeps the timestamp of the last `config.default` file it read in the configuration database files.) The `pconfig("RELOAD")` function will not modify any user-created variables or values, but will replace any default variables and values that are different than those in `config.default`.

The `pconfig("RELOAD")` function returns the NULL string and sets the PSL variable `errno` = 0 (E_PSL_NO_ERROR) if successful or a PSL error message string and nonzero `errno` return value otherwise. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

REPLACE Option

The `pconfig("REPLACE")` function replaces all existing values of *variable* with *value*:

- If *variable* does not exist, the `pconfig("REPLACE")` function creates *variable* and adds *value* to it.
- If *variable* exists and has no values, the `pconfig("REPLACE")` function adds *value* to *variable*.
- If *value* is already in the list of values for *variable*, the `pconfig("REPLACE")` function deletes all other values so that only *value* remains.

The `pconfig("REPLACE")` function returns a string containing the new value of the agent configuration variable and sets the PSL variable `errno` = 0 (E_PSL_NO_ERROR) when successful or a PSL error message string and nonzero `errno` return value otherwise. (See “pconfig() Function Messages and Return Codes” on page 1-260.)

BMC Software, Inc., Confidential and Proprietary Information

pconfig() Function Messages and Return Codes

The `pconfig()` function detects and reports any of several error conditions by returning text string messages and setting the PSL variable `errno`. The following table summarizes the relationship between the `errno` return codes and the `pconfig()` function messages. The topics following the table briefly describe the `pconfig()` function messages.

PSL errno variable		pconfig() return value
Value	**Message**	**Message**
0	E_PSL_NO_ERROR	NULL string (except for LIST and GET options)
96	E_PSL_BAD_FUNCTION_PARAMETER	Syntax Error: pconfig argument 1 missing Syntax Error: argument 1 is invalid Syntax error: pconfig argument 2 is missing Syntax error: pconfig argument 2 is not a valid variable name Syntax error: pconfig argument 5 is not a valid variable name
117	E_PSL_ACL_FAILED	Configuration: write access not allowed due to ACL
118	E_PSL_CONFIG_FAILED	Configuration: Database PURGE failed Configuration: Database RELOAD failed

Configuration: write access not allowed due to ACL

The `pconfig()` function returns this text string to indicate that it was unable to write to the PATROL Agent configuration file because of Access Control List (ACL) restrictions. That is, the account under which the `pconfig()` command executed does not have permission to write to the PATROL Agent configuration file. When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 117 (E_PSL_ACL_FAILED). The `pconfig()` function can return this message for the `APPEND`, `DELETE`, `MERGE`, `PURGE`, `RELOAD`, and `REPLACE` options.

BMC Software, Inc., Confidential and Proprietary Information

Configuration: Database PURGE failed

The `pconfig()` function returns this text string to indicate that it was unable to purge the PATROL Agent configuration file. (Either it could not remove the existing file, or it could not create the new file from the `config.default` file). When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 118 (E_PSL_PCONFIG_FAILED). The `pconfig()` function can return this message for the `PURGE` option.

Configuration: Database RELOAD failed

The `pconfig()` function returns this text string to indicate that it was unable to reload the PATROL Agent configuration file default values from the `config.default` file. (This can be a `config.default` file problem or a problem reading the configuration database files.) When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 118 (E_PSL_PCONFIG_FAILED). The `pconfig()` function can return this message for the `RELOAD` option.

NULL String

The `pconfig()` function returns the NULL string to indicate it successfully completed the operation. When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 0 (E_PSL_NO_ERROR). The `pconfig()` function returns this message for the `APPEND`, `DELETE`, `MERGE`, `PURGE`, `RELOAD`, and `REPLACE` options.

Syntax Error: pconfig argument 1 missing

The `pconfig()` function returns this text string to indicate that no option (`APPEND`, `DELETE`, `FORMAT`, `MERGE`, and so on) was specified. When the pconfig() function returns this message, it sets the PSL variable `errno` = 118 (E_PSL_PCONFIG_FAILED).

BMC Software, Inc., Confidential and Proprietary Information

Syntax Error: argument 1 is invalid

The `pconfig()` function returns this text string to indicate that the specified option was not one of `APPEND`, `DELETE`, `FORMAT`, `GET`, `LIST`, `MERGE`, `PURGE`, `RELOAD`, or `REPLACE`. When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 96 (E_PSL_BAD_FUNCTION_PARAMETER).

Syntax Error: pconfig argument 2 missing

The `pconfig()` function returns this text string to indicate that its option requires *variable* and *variable* was not specified. (The *`value`* is always optional and always defaults to the NULL string when omitted.) When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 96 (E_PSL_BAD_FUNCTION_PARAMETER). The `pconfig()` function can return this message for the `APPEND`, `DELETE`, `GET`, `MERGE`, and `REPLACE` options.

Syntax Error: pconfig argument 2 is not a valid variable name

The `pconfig()` function returns this text string to indicate that the *variable* parameter has an invalid syntax. When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 96 (E_PSL_BAD_FUNCTION_PARAMETER). The `pconfig()` function can return this message for the `APPEND`, `DELETE`, `GET`, `MERGE`, and `REPLACE` options.

Syntax Error: pconfig argument 5 is not a valid variable name

The `pconfig()` function returns this text string to indicate that the *variable* parameter has an invalid syntax. When the `pconfig()` function returns this message, it sets the PSL variable `errno` = 96 (E_PSL_BAD_FUNCTION_PARAMETER). The `pconfig()` function can return this message for the `FORMAT` option.

BMC Software, Inc., Confidential and Proprietary Information

Examples

The following examples show a couple of ways to use the `pconfig()` function.

pconfig() Function Options

The following PSL script contains examples of the `pconfig()` function and its different options:

```
# Get the value for a configuration variable
function InetVarGet( app, inst, var ) {
   local varpath;
   varpath = "/InetSetup/" . app . "/" . inst . "." . var;

   # Looking up the (non-default) configuration vars
   return( nthlinef(pconfig("GET", varpath), 2, "") );
}

# Set the value of a configuration variable
function InetVarSet( app, inst, var, val ) {
   local varpath;
   varpath = "/InetSetup/" . app . "/" . inst . "." . var;
   return( pconfig("REPLACE", varpath, val) );
}

# Delete a configuration variable
function InetVarDelete( var ) {
   return( pconfig("DELETE", var) );
}

# Get a list of configuration variables
function InetVarList( expr ) {
   local tlist, all_list;
   all_list = pconfig("LIST");
   tlist = grep("^/InetSetup/", all_list);

   if (expr) {
      tlist = grep( expr, tlist);
   }

   return( sort( tlist ) );
}
```

BMC Software, Inc., Confidential and Proprietary Information

Sample from the Make Current KM's Preloaded Menu Command

The following script uses the `pconfig()` function as part of the PSL script for a **Make Current KM's Preloaded** menu command:

```
#make_preloaded_km_list.psl
#
#
# History:

requires response_def_lib;
requires unix_misc_lib;
response_def();
from = get_console_info();
if(console_type() != 1) {
   error_pop-up("Only a Developer Console can set preloaded KM's");
   event_trigger("0", "41", "INFORMATION", 9, "Attempt to set preloadedKMs by ".from);
   exit;
}
buf = system("%DUMP KM_LIST");
buf = nthlinef(buf,"4-".(int(lines(buf)) - 1));
kms = "";
foreach lin (buf){
   kms = kms . ntharg(lin, 2) . ".km=1\n";
}
kms = sort(kms);
resp_ret = response("Make Current KM's Preloaded",-1,"h=430,w=395",
    LABEL."Select KM's to preload (default: all)",
    SEP_HORIZ,
    LIST_MULTIPLE.kms,
    RADIO_HORIZ."Merge into preload list=1\nOnly preload these=0\n"
    );
act = trim(nthlinef(resp_ret,1), "\n");
if(act){
   sel_list = trim(nthlinef(resp_ret,2), "\n");
   merge_rep = trim(nthlinef(resp_ret,3), "\n");
   work = "";
   foreach word sel_num (sel_list) {
      work = work . nthargf(nthlinef(kms, sel_num), 1, "=") . "\n";
   }
   maxlen = get("/maxConfigValLen");
   if(length(kms) > maxlen){
      error_pop-up("KM list too long, cannot quick-set");
      exit;
   }
   if(merge_rep == 1){
      method = "MERGE";
      yn = rusure("Merge these into preloaded KM list?\n\n".work);
   }
   else {
      method = "REPLACE";
      yn = rusure("Set preloaded KM list to:\n\n".work);
   }
   if(yn == 1){
      kms = ntharg(work, "1-", "\n", ",");
      event_trigger("0", "41", "INFORMATION", 9, "/AgentSetup/preloadedKMs changed by ".from);
      pconfig(method, "/AgentSetup/preloadedKMs", kms);
   }
}
```

BMC Software, Inc., Confidential and Proprietary Information

popen()

Open a PSL channel to a process

Format

```
popen(type,command,[instance],[username],
      [password])
```

Parameters

Parameter	Definition
type	command processor that interprets and executes *command* **Valid Values** the built-in command types OS or a valid user-defined command type **Note:** You cannot use PSL as the *type* parameter.
command	syntax of the submitted command
instance	optional application instance against which *command* executes **Default** the application instance that is the nearest ancestor of *command*
username	optional user name under which the process channel is opened **Default** the user name under which the `popen()` function is called
password	optional password under which the process channel is opened **Default** the password under which the `popen()` function is called

BMC Software, Inc., Confidential and Proprietary Information

Description

The `popen()` function spawns a process to execute a command of a defined type and returns a channel number which can then be used to read the command's output or write messages to the command. (Refer to the PSL `read()` and `write()` functions on pages 1-290 and 1-528 respectively.) The `popen()` function returns –1 on error.

The `popen()` function can be used to submit a command to run in the background. Sometimes a long-running process like a daemon may be better run in the background, but care should be taken when submitting commands in the background.

Normally, when a command is executed by a parameter, or as a recovery action, it is executed in the foreground, and the parameter waits for the command script to finish before refreshing. This ensures that the recovery action is complete before the parameter is checked to see if an additional recovery action is required.

However, if a recovery action is submitted in the background, the command script can finish before the recovery action is complete. When the command script finishes, the parameter refreshes. If the parameter is still in an alarm state, a second recovery action may start and conflict with the first recovery action.

Example

The following PSL script uses the popen() function to open a process channel to the Unix shell command interpreter and then uses the write() function to submit a script to the command interpreter.

```
chan = popen("OS","/bin/sh");
if (chan_exists(chan))
{
   write(chan,"/usr/local/myscript &");
   close(chan);
}
```

BMC Software, Inc., Confidential and Proprietary Information

poplines()

Return the specified number of lines from the beginning of a variable

Format

```
poplines(text_variable, num_lines, [w])
```

Parameter

Parameter	Definition
text_variable	variable to be edited by poplines() This parameter must be a variable. You cannot use a string because poplines() edits the content of the variable.
num_lines	number of lines to be returned by this function
w	optional flag that removes all trailing whitespace from the text returned by poplines()

Description

The poplines() function returns the specified number of lines from the beginning of the user-defined variable, and it deletes these lines from the variable.

This function returns NULL when it encounters an error. Use the following values for the *errno* variable to troubleshoot poplines().

BMC Software, Inc., Confidential and Proprietary Information

errno Value	Description of Failure
0	No error if poplines() returns NULL and errno equals zero, *text_variable* probably contains only white space that gets removed because of the w flag
68	E_PSL_EDOM *text_variable* is NULL
96	E_PSL_BAD_FUNCTION_PARAMETER *text_variable* is not a variable, *num_lines* is not a number, or *num_lines* is less than 1

Example

The following example demonstrates the poplines() function.

```
errno=0;
text_var=" Line 1\n Line 2\n Line 3\n Line 4";
print("Before poplines, text_var=".text_var);
result_text=poplines(text_var, 3);
print("\n\nAfter poplines, text_var=".text_var);
print("\n\nThe following lines were removed:\n". result_text);
```

The following output results from the preceding code.

```
Before poplines, text_var= Line 1
 Line 2
 Line 3
 Line 4

After poplines, text_var= Line 4

The following lines were removed:
 Line 1
 Line 2
 Line 3
```

BMC Software, Inc., Confidential and Proprietary Information

popup_report()

Pop up a task window on interested consoles

Format

```
popup_report(title,[application],[handle],
      [data])
```

Parameters

Parameter	Definition
title	the title that should appear on the task window
application	optional name of the application from which to determine which consoles to pop up the task window on
handle	optional task ID returned by a previous call to popup_report()
data	optional initial data to be displayed in the task window

Description

The `popup_report()` function pops up a task window on all consoles that are interested in the specified application. If *application* is the empty string, "", then the task window pops up on those consoles interested only in the PSL process that is making the call to the `popup_report()` function. The *handle* becomes invalid if all consoles have destroyed the task window, thereby removing all the status information about the task.

If *handle* is not the empty string, the following actions occur:

- If the report exists (that is, the task window has been popped up previously), then the task window pops up only on those consoles that have started monitoring the application/process since the last call to the `popup_report()` function.

BMC Software, Inc., Confidential and Proprietary Information

- The *data* does not necessarily go to all of the consoles that are interested in the task, because the consoles that were previously popped up already have the data.

The popup_report() function returns one of the following values:

- the task ID (that is, *handle*) if the function call is successful.
- the empty string, "", if the function call fails (that is, if an invalid *handle* is used).

Note

The `popup_report()` and `write_to_report()` functions use *handle* to identify the report to be popped up or written to, respectively.

To write text to the popped-up task window, see the write_to_report() function on page 1-530.

Examples

The following examples demonstrate how to pop up a task window on interested consoles. (Both examples call the `write_to_report()` function to write text to the popped-up windows.)

Simple Example

```
# Pop up a report on all connected Consoles
handle = popup_report("File System Report");
if (handle) {
   write_to_report(handle,system("/bin/df"));
}
```

Complex Example

```
# Pop up a report on all Consoles that are currently monitoring the
# FILESYSTEM application. And refresh the report every 10 seconds and ensure
# that new Consoles can also see the report.
handle = popup_report("File System Report","FILESYSTEM","",
       "Started at ".date()."\n");
while (handle) {
   sleep(10);
   # Pop up a report on all consoles that started monitoring the
   # FILESYSTEM application after the popup_report() function was called
   handle = popup_report("File System Report (already running)",
       "FILESYSTEM",handle,"You missed the start\n");
   write_to_report(handle,system("/bin/df"));
}
```

BMC Software, Inc., Confidential and Proprietary Information

pow()

Raise a number to a power

Format

`pow(`*`base`*`,`*`exponent`*`)`

Parameters

Parameter	Definition
base	numeric value that is to be multiplied by itself *exponent* number of times
exponent	numeric value that indicates the number of times *base* should be multiplied by itself The *exponent* must be positive if *base* = 0, and *exponent* must be an integer if *base* < 0

Description

The `pow()` function returns the value of *`base`* raised to the power *exponent*, or $base^{exponent}$.

The output range for the `pow()` function is $-\infty < \texttt{pow()} < \infty$.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL script uses the `pow()` function to produce a table of powers of the integers from one to nine.

```
function main() {
   print(" n     n^4        n^5        n^6        n^7        n^8        n^9\n");
   print("--  ---------  ---------  ---------  ---------  ---------  ---------\n");
   base = 1;
   while (base <= 9) {
      printf("%2d",base);
      exponent = 4;
      while (exponent <= 9) {
         printf("  %9d",pow(base,exponent));
         exponent++;
      }
      printf("\n");
      base++;
   }
}
```

The example produces the following table:

```
 n     n^4        n^5        n^6        n^7        n^8        n^9
--  ---------  ---------  ---------  ---------  ---------  ---------
 1          1          1          1          1          1          1
 2         16         32         64        128        256        512
 3         81        243        729       2187       6561      19683
 4        256       1024       4096      16384      65536     262144
 5        625       3125      15625      78125     390625    1953125
 6       1296       7776      46656     279936    1679616   10077696
 7       2401      16807     117649     823543    5764801   40353607
 8       4096      32768     262144    2097152   16777216  134217728
 9       6561      59049     531441    4782969   43046721  387420489
```

print()

Print text to a computer output window on the PATROL Console

Format

`print(`*`text1`*`,[...,`*`textn`*`])`

Parameter

Parameter	Definition
text1...textn	one or more text strings or variables containing a text string to be output by the `print()` function

Description

The `print()` function prints *text1*...*textn* to the computer system window on the PATROL Console. If the PSL script is a task, the output of the `print()` function is directed to the task's own window. The `print()` function always returns the NULL string.

Example

Each of the following `print()` functions displays the phrase "Hello World!" to the computer system window on the PATROL Console:

```
print ("Hello ","world!","\n");

print("Hello world!\n");

print("Hello ");
print("world!");
print("\n");
```

BMC Software, Inc., Confidential and Proprietary Information

printf()

Print text formatted to the C library `printf()` routine specification

Format

`printf(`*`format`*`)`

Parameter

Parameter	Definition
format	text, variable names, and control characters that specify the content and format of output to the computer or task output window: **Valid Values** %d signed decimal (identical to %i) %i signed decimal (identical to %d) %u unsigned decimal %o unsigned octal %x unsigned hexadecimal using `abcedf` %X unsigned decimal using ABCDEF %c unsigned character %s character string %e double precision form `drddde±ddd` where each d is a digit and `r` is the radix character %E double precision form `drdddE±ddd` where each d is a digit and `r` is the radix character %f decimal notation form `dddrddd` where each d is a digit and `r` is the radix character %g print in the style of %e if the exponent after conversion is less than –4, else print in style %f %G print in the style of %E with the precision specifying the number of significant digits %N Group digits into threes and separate with commas beginning at the right of the string %% print a % character The *format* parameter does not support the standard C pointer conversion characters %p and %n.
format (continued)	*The format* parameter permits the following flags: – left-justify and pad on the right with spaces + display plus sign when value is greater than zero 0 pad with zeros if no other padding is specified # alters the meaning of a conversion: •appends 0x or 0X to the %x and %X conversions •always appends the radix character to the %e, %E, %f, %g, and %G conversions •retains trailing zeros in the %g and %G conversions The # flag does not affect the %c, %d, %s, or %i conversions.

BMC Software, Inc., Confidential and Proprietary Information

Description

The `printf()` function displays output to the computer or task output window using formatting similar to the standard C `printf()` function.

A bad format or one that is valid for the C language but not for the `printf()` function results in a PSL run-time error that sets the PSL `errno` variable.

The `printf()` function return value is always the NULL string.

C Conventions Not Supported by the PSL printf Function

The `printf()` function does not support the C convention of using the asterisk (*) as a field width or precision indicator. The `printf()` function does not support the `%p` and `%n` conversion characters.

The length modifiers `h`, `l` (ell), and `L` are not valid and are ignored by the printf() function.

The `printf()` function format conversions are passed directly to the C library `printf()` routine on each platform. The output for obscure formatting features may differ across platforms, especially on platforms with libraries that do not conform to ANSI C requirements.

Conversion Differences between the C printf Routine and PSL printf Function

The format conversions have the same meaning between standard C and PSL, but the concept of variable types differs between the two.

PSL supports only string types for its variables, and thus string arguments to the `printf()` function are converted in a manner appropriate for the format conversion:

- Integral formats such as `%d` convert the string to signed integers.
- Noninteger numeric formats such as `%f` convert to floating point values.

BMC Software, Inc., Confidential and Proprietary Information

- `%c` prints the ASCII equivalent of its integer argument or for nonnumeric arguments the first character of its argument. (Applying `%c` to “65” will print ‘A’ and to “AB” will print ‘A’.)

- `%s` causes no conversion.

- `%%` requires no argument.

%N Format Conversion

The `printf()` function provides one nonstandard C extension—the `%N` conversion. The `%N` conversion preprocesses a numeric string so that commas separate each group of three digits beginning at the right side of the string.

For example, the `%N` conversion causes the following conversions:

1234 ⇒ 1,234
12345 ⇒ 12,345
123456 ⇒ 123,456

The `%N` conversions ignores initial minus signs and blanks while searching for the first sequence of digits so that `%N` can be applied to negative values. If no digits are found after the skipped characters, the printed argument is unchanged.

The `%N` conversion only modifies the first sequence of digits. For example, the `%N` conversion changes floating point numbers like 1234.1234 to become 1,234.1234 without changing to the digit sequence to the right of the decimal point.

BMC Software, Inc., Confidential and Proprietary Information

As part of the `%N` conversion, the `printf()` function performs a `%s` conversion using the field width and precision specifiers supplied in *`format`*. The `printf()` function prints the string resulting from the combined `%N` and `%s` conversions. Because of the embedded `%s` conversion, field width and precision under `%N` conversion have the same effect as with `%s`.

Note

Currently, no localization is supported by `%N`, and so the formatting achieved by `%N` does not change in different locales.

Example

The following series of printf() function calls illustrate several formatting characters used for numbers and strings:

```
string = "1257384.00234600000";
printf("          string as an integer : %d\n",string);
printf(" string as a formatted integer : %7N\n",int(string));
printf("              string as a real : %f\n",string);
printf("    string as a formatted real : %12.4f\n",string);
printf("  string as a character string : %s\n",string);
printf(" string in scientific notation : %e\n",string);
```

The example produces the following output:

```
          string as an integer : 1257384
 string as a formatted integer : 1,257,384
              string as a real : 1257384.002346
    string as a formatted real : 1257384.0023
  string as a character string : 1257384.00234600000
 string in scientific notation : 1.257384e+06
```

BMC Software, Inc., Confidential and Proprietary Information

proc_exists()

Verify that a process exists

Format

`proc_exists(`*`pid`*`)`

Parameter

Parameter	Definition
pid	process identifier number of the process whose existence is being verified

Description

The `proc_exists()` function returns TRUE if the process with process identifier *pid* exists, FALSE if it does not.

Example

The following example uses the proc_exists() function to verify that a process with a specific process ID exists:

```
if (proc_exists(1)) {
   print("A process with PID 1 is running\n");
}
```

BMC Software, Inc., Confidential and Proprietary Information

process()

Return a list of processes from the PATROL Agent process cache

Format

`process(`*`regular-expression`*`)`

Parameter

Parameter	Definition
regular-expression	character sequence that defines the pattern the `process()` function searches for in the PATROL Agent process cache The *regular-expression* conforms to the regular expressions defined in the Unix ed(1) command description and the Unix regexp(5) description. Following is a brief summary of several regular expression characters: ^ beginning of line $ end of line . match any single character * match zero or more repetitions of the preceding [] match any of the characters contained within [^] match any characters except those contained within

Description

The `process()` function returns the list of processes from the name and cmd columns in the PatrolAgent's process cache that match the regular expression *regular-expression*.

BMC Software, Inc., Confidential and Proprietary Information

Each entry in the list is a string formatted as follows:

```
pid  ppid   user   status   size   cputime   command_name   command-line
```

Parameter	Definition
pid	process identifier number
ppid	parent process identifier number
user	user name to which the process belongs
status	process status within the system **Valid Values** 0 nonexistent S sleeping W waiting R running I intermediate Z terminated T stopped X growing
size	process core image size Depending on the platform of the PATROL Agent, this measurement can be in blocks or kilobytes.
cputime	integer number of CPU seconds consumed by the process
command_name	first word of the command line that started the process
command-line	complete command line that started the process Note that the command line may have been modified during process execution.

Note

Some platforms do not support all the return values. For a specific platform, the `process()` function generally returns the same information as the `ps` command (`patrolps` command for OS/2).

The `process()` function returns the NULL string if no processes match *regular-expression*. The `process()` function returns a question mark (?) for any values that cannot be determined.

Example

The following example lists all active processes:

```
# find all processes
procs = process("");
print (procs);
```

The example produces the following output:

```
0      -1     <?>          ?  0         0         Idle  Idle
2      -1     <?>          ?  1616      351       System  System
21     2      SYSTEM       ?  8772      0         smss  smss
26     -1     <?>          ?  49256     0         csrss  csrss
35     21     SYSTEM       ?  40556     1         winlogon  winlogon
41     35     SYSTEM       ?  46876     99        services  services
44     35     SYSTEM       ?  38380     173       lsass  lsass
68     41     SYSTEM       ?  71236     7         spoolss  spoolss
96     -1     <?>          ?  41676     0         AMGRSRVC  AMGRSRVC
100    41     SYSTEM       ?  16860     0         inetd32  inetd32
114    41     SYSTEM       ?  29972     4         RpcSs  RpcSs
122    41     SYSTEM       ?  23052     0         snmp  snmp
147    35     pfarrell     ?  15468     0         nddeagnt  nddeagnt
158    -1     Agent        ?  18236     4         snmpmagt  snmpmagt
160    -1     pfarrell     ?  67408     567       Explorer  Explorer
127    160    pfarrell     ?  14596     0         SysTray  SysTray
162    160    pfarrell     ?  30540     0         ShStat  ShStat
153    160    pfarrell     ?  51384     73        HPPROPTY  HPPROPTY
64     160    pfarrell     ?  40144     0         HPNRA  HPNRA
161    160    pfarrell     ?  45896     283       FINDFAST  FINDFAST
199    -1     pfarrell     ?  24280     0         ddhelp  ddhelp
183    160    pfarrell     ?  87848     437       frame  frame
192    160    pfarrell     ?  37828     1984      PatrolAgent
82     192    pfarrell     ?  18168     44        PatrolPerf
105    160    pfarrell     ?  20364     0         DsServer  DsServer
213    160    pfarrell     ?  50292     799       Patrol  Patrol
246    192    Agent        ?  17008     0         cmd  cmd
222    160    pfarrell     ?  68308     343       ntvdm  ntvdm
145    -1     pfarrell     ?  36780     0         vantiv32  vantiv32
70     160    pfarrell     ?  25408     192       TICKER  TICKER
264    -1     <?>          ?  39548     0         wordview  wordview
238    160    pfarrell     ?  20784     0         cdplayer  cdplayer
140    70     pfarrell     ?  62092     92        NETSCAPE  NETSCAPE
0      -1     <?>          ?  1160632   0         _Total  _Total
```

BMC Software, Inc., Confidential and Proprietary Information

PslExecute()

Start a PSL process

Format

```
PslExecute(name,PslScript,[errorlist],
      [warninglist])
```

Parameters

Parameter	Definition
name	text string that provides a descriptive identifier for the new PSL process
PslScript	text string containing the PSL commands that should be executed
errorlist	optional variable to which the PSL interpreter/compiler will return text string error messages created while compiling *PslScript* **Default** the `PslExecute()` function does not return compilation error messages
warninglist	optional variable to which the PSL interpreter/compiler will return text string warning messages generated while compiling *PslScript* **Default** the `PslExecute()` function does not return compilation warning messages.

Description

The `PslExecute()` function compiles and executes the statements in *PslScript* and runs them in the PATROL Agent as the PSL script *name*. If *errorlist* and *warninglist* are specified, the PSL interpreter/compiler will return any error or warning messages that it generates to those variables.

BMC Software, Inc., Confidential and Proprietary Information

The `PslExecute()` function returns the following values:

- 0 if *name* has been successfully created and is executing
- 1 if *name* has experienced a compilation errors
- 2 if *name* has experienced a process creation error

Non-Blocking Feature

The PslExecute() function works with a parent and child PSL script. The parent script is the code that executes PslExecute(), and the child script is executed by PslExecute(). The child script is also known as the *PslScript* parameter.

The non-blocking feature is that the child script does not block the progress of the parent script. Another way to describe this feature is that the parent script does not wait for the child script to finish its work.

Example

The following example uses the `PslExecute()` function to create the PSL process called `myscript` to print the message `Hello world:`

```
PslExecute("myscript","print(\"Hello world\\n\");",errors,warnings);
printf("compilation errors for myscript :\n%s\n",errors);
printf("compilation warnings for myscript :\n%s\n",warnings);
```

The following example shows the non-blocking feature (“Non-Blocking Feature” on page 1-285) of PslExecute():

```
print("1\n");
PslExecute("pkolTest","print(\"2\\n\");",x,y);
print("3\n");
```

The following output is the result of the previous example:

```
1
3
2
```

BMC Software, Inc., Confidential and Proprietary Information

The previous output is nonsequential because of the non-blocking feature of PslExecute().

BMC Software, Inc., Confidential and Proprietary Information

PslSetOptions()

Set the run-time options for the calling PSL process to the specified value

Format

`PslSetOptions([`*`options`*`])`

Parameters

Parameter	Definition
options	integer value that specifies the options **Valid Values** 0 No options 1 Calculate the message digest 2 Send the current PSL source line in each XPC call These options can be selected together: 3 selects both 1 and 2.

Description

Use the `PslSetOptions()` function to set the run-time options for the calling PSL process to the specified value. The current value of the options mask is always returned. If no argument is given, then the process options are not changed; this is useful if you just want to retrieve the current options mask. This function is useful only if the PSL script uses the XPC (external PSL call) server.

Example

The following is an example of the `PslSetOptions()` function:

```
current_options = PslSetOptions(2);
```

BMC Software, Inc., Confidential and Proprietary Information

random()

Return a random number

Format

`random([`*`maximum`*`])`

Parameter

Parameter	Definition
maximum	optional upper bound for the values returned by random $0 \leq$ `random()` $\leq$ *maximum* – 1 **Valid Values** *maximum* > 0 **Default** *maximum* = 2^{32} – 1 (from the underlying C function)

Description

The `random()` function is equivalent to the standard C library `random()` function.

If *maximum* is zero or negative, the `random()` function will return a run-time error message.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example tests the random() function by simulating the rolling of a single six-sided die. If the die (and the random() function) are honest, each of the numbers is equally likely, and after an arbitrarily large number of throws, each will come up an approximately equal number of times:

```
function main() {
   srandom(time()); # seed random() with the system clock
   ones = twos = threes = fours = fives = sixes = 0;
   limit = 6000;
   count = 0;
   while (count++ <= limit) {
      roll = int(random(6)) + 1;
      switch (roll) {
         case 1: {ones++;}
         case 2: {twos++;}
         case 3: {threes++;}
         case 4: {fours++;}
         case 5: {fives++;}
         case 6: {sixes++;}
      }
   }
   expected = int(limit / 6);
   print("    expected    actual\n");
   print("    --------   --------\n");
   printf("%2d  %8d  %8d\n",1,expected,ones);
   printf("%2d  %8d  %8d\n",2,expected,twos);
   printf("%2d  %8d  %8d\n",3,expected,threes);
   printf("%2d  %8d  %8d\n",4,expected,fours);
   printf("%2d  %8d  %8d\n",5,expected,fives);
   printf("%2d  %8d  %8d\n",6,expected,sixes);
}
```

The example produces the following output:

```
    expected    actual
    --------   --------
 1      1000       984
 2      1000       977
 3      1000      1044
 4      1000       988
 5      1000      1036
 6      1000       971
```

BMC Software, Inc., Confidential and Proprietary Information

read()

Read from a PSL file or process channel

Format

`read(`*`channel`*`,[`*`size`*`])`

Parameters

Parameter	Definition
channel	the process I/O channel number from which the read() function reads data
size	optional integer value controlling the amount of data that the read() function reads from *channel* **Valid Values** • *size* = 0 read() returns as soon as it has read something from the channel. • *size* = –1 read() returns all data available by either reading until the end of file on a channel opened with `fopen()`, or by blocking until the child process completes for a channel opened with `popen()`. • *size* = *n*, where *n* is a number greater than zero read() returns *n* bytes from the output of the channel. As with -1, read() operates in blocking mode. • *size* = “n”, where “n” is a literal switch read() performs a non-blocking read of all data available and returns control immediately. If no text is available, the empty string is returned and the errno variable is not set. The non-blocking mode only applies to channels opened by `popen()`—it is ignored without error for channels opened by `fopen()`. **Note:** Do not compare the output to EOF. Use the chan_exists() function to determine if the channel is still open. See **Example** section. **Default** *size* = 0

BMC Software, Inc., Confidential and Proprietary Information

Description

The read() function returns the data it reads one byte at a time from *channel*. The read() function returns the value EOF (that is, the NULL string) on an end-of-file or error condition.

Channels are created by calling the `fopen()` or `popen()` function. (For more information about the `fopen()` function, see page 1-166. For more information about the `popen()` function, see page 1-265.)

Note

The read() function can block for a process channel created using the `popen()` function but not for a file channel created using the `fopen()` function.

To enforce serialization for shared channels, no two reader processes (that is, read() or `readln()` functions) can be blocked on the same channel. The second reader process that attempts to block on the shared channel will fail, returning the NULL string and setting the PSL variable `errno` to E_PSL_BUSY_CHANNEL.

Another possible shared channel failure can be caused by a `close()` function being executed against a channel that also has a blocked reader process. The `close()` function will cause the reader process to return the NULL string and set `errno` to E_PSL_UNBLOCKED_BY_CLOSE.

Issues for Internationalized PSL Scripts

The issues in this section require that you understand how to create Internationalized PSL scripts. For more information about Internationalized PSL scripts, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

BMC Software, Inc., Confidential and Proprietary Information

Codeset Conversion

When you use the popen() or fopen() function to open a channel, the codeset of the CODECVT locale category becomes the registered codeset of the channel. If the registered codeset differs from the CTYPE locale category, read() converts the codeset and returns text in the CTYPE codeset.

Size Parameter and Multi-byte Characters

The *size* parameter specifies the number of bytes that the function reads from the channel. Do not use this parameter if you expect to read multi-byte characters. Using *size* with multi-byte characters could cause read() to read only part of a multi-byte character. Also consider using the readln() function, which poses no risk of truncating multi-byte characters.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL example shows a blocking use of the read() function. This example opens a channel to the Windows operating system, executes a Windows `dir` command, and prints the directory entries returned by the `dir` command.

```
chan = popen ("OS", "dir");
data = read (chan,"-1");
print (data);
close (chan);
```

This example uses the read() function in non-blocking mode.

```
# This PSL script connects to an ftp server or times out after 10 seconds
# if it cannot connect.
IP = "ftp.bmc.com";
timeout = 10;

FTPChan = popen("OS", "ftp -nv ".IP);
timeouttime = time()+timeout;

do {
# The sleep stops this script consuming large amounts of CPU time
# due to looping whilst waiting for data
  sleep(1);
  data = read(FTPChan,"n");
} until(!chan_exists(FTPChan) || data!="" || (time()>=timeouttime));

# Either some data has been read or the timeout has expired.
if(!chan_exists(FTPChan) || data==="") {
  # Tidy up
  if(chan_exists(FTPChan)) {
    close(FTPChan);
  }
  print("Connection failed!\n");
} else {
  DataRead = DataRead.data;
  print(DataRead);

  # Carry on reading/writing to the channel. Data was returned.
  print("Data returned on the channel\n");
}
```

BMC Software, Inc., Confidential and Proprietary Information

readln()

Read a line of data from a PSL file or process channel

Format

`readln(`*`channel`*`)`

Parameter

Parameter	Definition
channel	the process I/O channel number from which the readln() function is to read data

Description

The `readln()` function reads the next line of data from *channel* and returns it. The `readln()` function returns the value EOF (NULL) on end-of-file or error and sets `errno` = 55 (E_PSL_READ_FAILED) on end-of-file.

Channels are created by calling the `fopen()` or `popen()` function. (Refer to the `fopen()` or `popen()` functions on pages 1-166 or 1-265 respectively.)

Note

The `readln()` function can block for a pipe channel created using the `popen()` function but not for a file channel created using the `fopen()` function.

To enforce serialization for shared channels, no two reader processes (that is, `read()` or `readln()` functions) can be blocked on the same channel. The second reader process that attempts to block on the shared channel will fail, returning the NULL string and setting the PSL variable `errno` to E_PSL_BUSY_CHANNEL.

BMC Software, Inc., Confidential and Proprietary Information

Another possible shared channel failure can be caused by a `close()` function being executed against a channel that also has a blocked reader process.The `close()` function will cause the reader process to return the NULL string and set `errno` to E_PSL_UNBLOCKED_BY_CLOSE.

Limitation

The `readln()` function has a line limitation of 4K when executed against files opened with the `fopen()` function. The `readln()` function may truncate lines longer than 4K. This limitation does not apply to channels opened using the `popen()` function.

Example

The following example reads lines of text from a file or process channel:

```
chan=fopen("/usr/users/passwd.dat","r");
if (chan_exists(chan)) {
    data=readln(chan);
#The readln() function sets errno = 55 (E_PSL_READ_FAILED) on end-of-file
    while ( errno != 55) {
        printf("%s\n",data);
        data=readln(chan);
    }
    close(chan);
}
```

BMC Software, Inc., Confidential and Proprietary Information

refresh_parameters()

Refresh PATROL parameters

Format

`refresh_parameters(`*`object`*`,[GLOBAL])`

Parameters

Parameter	Definition
object	name of the PATROL object that is to be updated
`GLOBAL`	mnemonic that instructs the `refresh_parameters()` function to update all instances of *object* if *object* is a global parameter

Description

The refresh_parameters() function runs the PATROL parameters identified by *object*:

- If *object* is a computer or application instance, run all parameters that belong to *object.*
- If *object* is a local parameter, run the parameter for *object.*
- If *object* is a global parameter and "GLOBAL" is specified, run all instances of the global parameter *object.*
- If *object* is a global parameter and "GLOBAL" is *not* specified, run only the named instance of *object.*

To properly refresh a consumer parameter, you must also refresh its collector parameter.

BMC Software, Inc., Confidential and Proprietary Information

Examples

The following refresh_parameters() function refreshes the PAUserErrors parameter for the instance PATROLAGENT:

```
refresh_parameters("/PATROLAGENT/PATROLAGENT/PAUserErrors");
```

The following refresh_parameters() function refreshes all parameters for the instance PATROLAGENT:

```
refresh_parameters("/PATROLAGENT/PATROLAGENT");
```

BMC Software, Inc., Confidential and Proprietary Information

remote_close()

Close a session with a remote PATROL Agent

Format

`remote_close(sessionID)`

Parameter

Parameter	Definition
sessionID	the identifier of the PATROL Agent session that is to be closed The *sessionID* is returned by the `remote_open()` function when you open a remote PATROL Agent session.

Description

The remote_close() function closes the connection to the remote PATROL Agent that is identified by *sessionID*. The remote_close() returns 1 if it successfully closes the remote session, and 0 otherwise.

See “remote_open()” on page 1-315 for more information on using the remote functions.

Example

The following example shows the remote_close() function:

```
if (remote_close(session) != 1) {
   # call user-written error handling routine
   CloseRemoteFailure();
}
```

remote_event_query()

Return a list of events in the PATROL Event Manager (PEM) repository of a remote PATROL Agent that match specified filter criteria

Format

```
remote_event_query(sessionID,[maxcount],
      [timeout],[delimiter],[format],
      [start-time],[stop-time],[status],
      [type],[node],[origin],[pattern],[IDrange],
      [class],[severity])
```

Parameters

Parameter	Definition
sessionID	the identifier of the remote PATROL Agent session to which the remote event query is directed The *sessionID* is returned by the `remote_open()` function when you open a remote PATROL Agent session.
maxcount	string that is the maximum number of events that will be returned in the event query Specifying the NULL string (" ") causes *maxcount* to default to 100.
timeout	string that is the maximum time in minutes that the session is willing to wait for the results of the remote event query

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
delimiter	string that is used to separate each event in the list of events returned by the remote event query **Valid Values** • " " indicating that a newline character \n will separate the entries • any valid characters including PSL string literals (see *PATROL Script Language Reference Manual Volume 1—PSL Essentials*, “PSL String Literals”)
format	format string used to present each event entry See “Specifying the event_query() Output Format” on page 1-124 **Default** " " is equivalent to the string "%s %s %s %s %s %s %s %s\n" where the eight strings returned are (in order): • event ID assigned by the PEM • event status • event type • event timestamp • host name that produced the event • application class or instance that produced the event • text string from the event description field • text string from the event diary field

Parameter	Definition
Filter:	
start-time	time endpoint that specifies the oldest event timestamp that is valid for the remote event query **Valid Values** • "" indicating January 1, 1970 at 00:00:00 • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The *start-time* variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format, the current year is used when *yy* is omitted.
stop-time	time endpoint that specifies the most recent event timestamp that is valid for the remote event query **Valid Values** • "" indicating January 1, 1970 at 00:00:00 • PSL backward compatible: *MMddhhmm*[*yy*] • RFC-822: *day date month year hh:mm:ss* • Unix: *day month date hh:mm:ss year* • PSL date(): *day month date hh:mm:ss year* The *start-time* variables have the following values: *day*—Sun Mon Tue Wed Thu Fri Sat *MM*—01 to 12 *month*—Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec *dd* or *date*—01 to 31 *hh*—00 to 23 *mm* and *ss*—00 to 59 *year*—1902 to 2037 [*yy*]—00 to 99 (19*yy* is assumed) In the PSL compatibility format, the current year is used when *yy* is omitted.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
status	event status that is valid for the remote event query **Valid Values** O OPEN A ACKNOWLEDGED C CLOSED E ESCALATED D DELETED **Examples** • "O,A,C,D" matches all statuses except ESCALATED • "O,A,C,E,D" or " " matches all statuses • "O,C" matches only statuses OPEN and CLOSED
type	event type that is valid for the remote event query **Valid Values** I INFORMATION S CHANGE_STATUS E ERROR W WARNING A ALARM R RESPONSE **Examples** • "S,E,W,A,R" matches all event types except INFORMATION • "I,S,E,W,A,R" or " " matches all event types • "W,A" matches only event types WARNING and ALARM
node	computer system name that is valid for the remote event query **Valid Values** " " indicating all computer systems listed in the PEM repository or a host name character string
origin	application instance or class name that is valid for the remote event query **Valid Values** " " for all application classes or a character string indicating a specific application instance or class

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
pattern	character string within the event description field that is valid for the remote event query **Valid Values** " " indicating any characters or a character string
IDrange	string that defines the range of PATROL event IDs that are valid for the remote event query **Valid Values** x event ID x x/y event IDs between and including x and y -/y event IDs less than and including y x/- event IDs greater than and including x -/- all events where x and y are integers such that $0 \leq x \leq y \leq 2{,}147{,}483{,}647$
class	event class that is valid for the remote event query **Valid Values** " " indicating all event classes or a character string specifying a specific event class
severity	string containing the integer event severity that is valid for the remote event query **Valid Values** " " for all event severities or an integer from 1–5 (5 being most severe) indicating a minimum severity

Note

When the two digit year, *yy*, is specified with the PSL compatible time format for the *start-time* and *stop-time* parameters, the function assumes the year is 19*yy*. You should either omit the year when using this format, or use a different date format if you need to specify the year.

BMC Software, Inc., Confidential and Proprietary Information

Description

The remote_event_query() function returns a list of up to *maxcount* events found in the remote PATROL Agent Event Manager repository that match the filter criteria. The session to the remote PATROL Agent is identified by *sessionID*. The returned events are formatted as specified by *format*.

The remote_event_query() function returns the NULL string if no events were found in the event repository that match the filter criteria.

Specifying the remote_event_query() Output Format

The remote_event_query() function *format* parameter is similar to the specification string used for the standard C library printf() function. The *format* parameter can contain alphanumeric characters for use as titles and field names, and string literals for spacing, tabbing, and carriage control.

PATROL macro variables within the *format* parameter identify the fields that the remote_event_query() function returns. Table 1-7, “event_query() Function Macro Variables,” on page 1-124, describes the macro variables available to the remote_event_query() function.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the remote_event_query() function:

```
print(
   "The result of the remote query\n",
   remote_event_query(
      mysession,  # session handle returned by remote_open
      "100",      # return a maximum of 100 events
      "1",        # call will time out in one minute
      "\n\n",     # separate each event with two newline characters
      "event ID: %{EV_ID}\nevent type: %{EV_TYPE}\ndescription: %{EV_DESC}",
                  # event output format
      "",         #   event filter: any start time
      "",         #   event filter: any end time
      "O",        #   event filter: only OPEN status
      "A,W",      #   event filter: only ALARM AND WARNING states
      "",         #   event filter: any node
      "ORACLE",   #   event filter: ORACLE as an originating application
      "",         #   event filter: any pattern in the event description
      "-/-"       #   event filter: all events regardless of ID range
      ""          #   event filter: any event class
   )
   . "\n"
);
```

BMC Software, Inc., Confidential and Proprietary Information

remote_event_trigger()

Create an event instance for the PATROL Event Manager of a remote PATROL Agent

Format

```
remote_event_trigger(sessionID,origin,catalog,
      class,type,severity,[argument1,...argument20])
```

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
sessionID	the identifier of the remote PATROL Agent session to which the remote event trigger is directed The *sessionID* is returned by the `remote_open()` function when you open a remote PATROL Agent session.
origin	character string indicating the application instance or class that originated the event
catalog	character string name of the PATROL catalog to which the event belongs
class	character string name of the event in the class to which the event belongs
type	the event type as displayed in the PATROL Event Manager **Valid Values** ALARM WARNING ERROR CHANGE_STATUS INFORMATION
severity	integer value indicating the severity of the event **Valid Values** 0–5 (5 is most severe). A severity of 0 will send an event to the PATROL Agent, but the event will not appear in the PATROL Event Manager.
argumentn	optional arguments passed to the PATROL Event Manager and displayed in the event detailed report This function permits a maximum of 20 arguments.

Description

The remote_event_trigger() function creates an event instance that appears in the PATROL Event Manager of a remote PATROL Agent

BMC Software, Inc., Confidential and Proprietary Information

identified by *sessionID*. The origin of the event instance is *origin*. The remote_event_trigger() function assumes that the event catalog and class are already defined in the PATROL Event Manager for the remote PATROL Agent. The remote_event_trigger() function returns a 1 for successful execution and a 0 for failure.

The remote_event_trigger() function is different than the event_trigger() function because the remote_event_trigger() function will not print output to the PATROL console. Therefore, when you use the remote_event_trigger() function, it may appear that notification commands are not executing because they do not appear in the console window.

Note

The `remote_event_trigger()` function returns immediately after initiating the process. It does not wait for completion of the trigger. Therefore, if you use the `remote_event_query()` function immediately after the `remote_event_trigger()` function, the event may not yet exist on the remote host. You may want to use the `sleep()` function as a delay between the `remote_event_trigger()` and `remote_event_query()` functions when they are used together.

Example

The following is an example of the remote_event_trigger() function:

```
remote_event_trigger(
   mysession,   # session handle returned by remote_open()
   "myorigin",  # event origin: myorigin
   "STD",       # event catalog: STANDARD
   "41",        # event class: 41
   "ALARM"      # event type: ALARM
   "5",         # severity: 5 (most severe)
   "myarg"      # dynamic argument 1
);
```

remote_file_send()

Send an ASCII file to a remote PATROL Agent

Format

```
remote_file_send(sessionID,srcfilename,
      destfilename,[catalog],[class],sendID,
      [access])
```

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
sessionID	the identifier of the remote PATROL Agent session to which the file is sent The *sessionID* is returned by the `remote_open()` function when you open a remote PATROL Agent session.
srcfilename	full path name of the ASCII file that is to be sent to the remote PATROL Agent
destfilename	name to be given the ASCII file on the remote PATROL Agent host The file is written to the path: **$PATROL_REMOTE/destfilename**
catalog	name of the catalog to which the event generated by the file transfer belongs **Valid Values** a PATROL event catalog name or `""` indicating the `STANDARD` catalog.
class	name of the class to which the event generated by the file transfer belongs **Valid Values** a PATROL event class name or `""` indicating the `RemProcess` class.
sendID	user-defined identifier for the file send operation

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
access	integer designating the access mode assigned to the destination file In Unix, file access is defined using an octal representation of the binary rwxrwxrwx (read, write, execute) permissions for owner, group, and world. The access parameter is the decimal equivalent of the octal value used in Unix. To define the *access* parameter, determine the octal value for the file permissions and convert it to a decimal value. **Valid Values** 000-777 (octal) **Default** 666 (octal) 438 (decimal) which gives read and write permission to owner, group, and world See “Determining the Decimal value for access” section for more information on determining the value for the *access* parameter.

Description

The remote_file_send() function copies *srcfilename* from the host on which the function executes to *destfiename* on the PATROL Agent host identified by *sessionID*. The remote_file_send() function returns 1 if you called this function with valid parameters. Otherwise the function returns 0. There is no built-in limit to the size of the file that can be transferred.

If the PATROL_REMOTE environment variable is defined for the remote host, the file is written to **$PATROL_REMOTE**. If the PATROL_REMOTE environment variable is not defined, the file gets written to **$PATROL_HOME/remote** on the remote PATROL Agent.

The remote_file_send() function generates an event on the PATROL Agent host identified by *sessionID* for the file send operation using the *catalog* and *class* parameters. If the *catalog* and *class* parameters are omitted, the remote_file_send() function creates the STANDARD catalog event RemProcess.

BMC Software, Inc., Confidential and Proprietary Information

See “remote_open()” on page 1-315 for more information on using the remote functions.

Determining the Decimal value for *access*

The *access* parameter is the decimal equivalent of the octal representation of the binary rwxrwxrwx (read, write, execute) permissions for owner, group, and world used under Unix.

The simplest way to determine the decimal equivalent is to determine the octal value for Unix and convert it to a decimal value.

Octal Permissions in Unix

In Unix, the valid range for the file permissions is 000–777 where the first digit is the owner permission, the second digit is the group permission, and the third digit is the world permission.

The permissions are assigned to octal values as follows:

0 = no access (---)
1 = execute(--x)
2 = write(-w-)
3 = write and execute(-wx)
4 = read(r--)
5 = read and execute(r-x)
6 = read and write(rw-)
7 = read, write, and execute (rwx)

For Example, 666 is the octal value that would set the Unix file permissions to rw-rw-rw- which is read and write permission to the file for owner, group, and world.

Decimal Permissions

The octal values for permissions are easy to determine. To determine the *access* parameter for the remote_file_send() function, start with the octal value and convert it to a decimal value. See “convert_base()” on page 1-65 for information on using the PSL `convert_base()` function to convert octal to decimal.

Table 1-10 compares the decimal and octal values for common file permissions.

Table 1-10 Decimal and Octal File Permissions

Octal	Decimal	Permissions (Owner, Group, World)
444	292	r--r--r--
664	436	rw-rw-r--
666	438	rw-rw-rw-
744	484	rwxr--r--
770	504	rwxrwx---
777	511	rwxrwxrwx

For more information on Unix file permissions, see the `chmod(1)` user command in your Unix documentation.

Example

The following is an example of the remote_file_send() function where the *access* parameter is set as an octal value of 744 (read, write, and execute access for the owner and read access for the group and world–rwxr--r--) and then converted to a decimal value using the convert_base() function:

```
# set the file permission as an octal value of 744 for rwxr--r--
octal_access=”744”;

# convert the octal permission to a decimal for the remote_file _send()
decimal_access=convert_base(octal_access,8,10);

remote_file_send(
   mysession,       # session handle returned by the remote_open() function
   "/tmp/myfile",   # name of file to send
   "myremotefile",  # name of file on remote PATROL Agent host
   "",              # STANDARD catalog for file send event
   "",              # RemProcess class for file send event
   "myfileID",      # myFileID identifies this file send operation
   decimal_access   # file access permission of rwxr--r--
);
```

BMC Software, Inc., Confidential and Proprietary Information

remote_open()

Open a communication session with a remote PATROL Agent

Format

```
remote_open(host,port,[account-mode],[username],
      [password],[options])
```

Parameters

Parameter	Definition
host	host name on which the remote PATROL Agent is running
port	UDP or TCP/IP port number on which the remote PATROL Agent is listening
account-mode	optional account information that is used to open the session with the remote PATROL Agent **Valid Values** • `""` or omitted—ignore *username* and *password* and use the following account information: the nearest instance the parent of the nearest instance the application to which the nearest instance belongs • `"NONE"`—use the unencrypted username and password • `"pem"`—encrypt username and password with Privacy-Enhanced Mail (PEM) • `"des"`—encrypt username and password with triple data encryption standards-cipher block chain (DES-CBC) **Note:** For more information about encryption, see PATROL Security User Guide.
username	username that is required to open a session with the remote PATROL Agent

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
password	password that is required to open a session with the remote PATROL Agent. The *password* may or may not be encrypted
options	a comma delimited list of options in the following format: "`protocol=UDP,timeout=nnn,retries=nnn,heartbeat=nnn`" • protocol—the default protocol for this function is UDP, but it can be changed to TCP with the *options* parameter The following options only apply to the UDP protocol: • timeout—the maximum time in seconds that the function waits for the communication session to start • retries—the maximum number of times that the function tries to establish a communication session • heartbeat—the number of seconds between attempts to keep the UDP connection open

Description

The remote_open() function opens a communication session between the PATROL Agent on which it is running and the PATROL Agent identified by *host* and *port*, using UDP unless TCP is specified in the *options* parameter. If *account* = "NONE", the remote_open() function uses the unencrypted *username* and *password* to open the session; both must be valid on the remote PATROL Agent. If *account* = "" or is omitted, the remote_open() function uses the account information from the nearest instance, instance parent, or application.

BMC Software, Inc., Confidential and Proprietary Information

The remote_open() function returns the session ID if successful; otherwise, it returns the NULL string.

Note

If you are using the remote functions across a firewall, you must set the PATROL Agent's `/AgentSetup/localPortForRemoteOpen` configuration variable to the local UPD port number that will be used for the `remote_open()` function. This port must be different than the PATROL Agent's port if you are sending files across a firewall. See the *PATROL Agent Reference Manual* for more information on setting PATROL Agent configuration variables.

Using the Remote File Functions

The remote file functions consist of the following PSL built-in functions:

- remote_open()
- remote_file_send()
- remote_close()

To effectively use these functions, it is important to understand how these functions work together.

Understanding Remote Functions

The `remote_open()` function connects to a remote agent, supplies authentication information, and returns immediately. It does not wait for the authentication to take place. The user must ensure that the username/password information being supplied in the `remote_open()` function is a valid username/password on the remote PATROL Agent. An invalid username/password causes the remote PATROL Agent to reject the session ID. This can shut down an operation before it is complete. Since the `remote_open()` function can return a session ID when the authentication fails, there is no guarantee that the session ID can be used to communicate with the remote agent.

BMC Software, Inc., Confidential and Proprietary Information

The `remote_file_send()` function transfers an ASCII file to a remote PATROL Agent. However, the function only starts the operation and returns immediately. The actual file transfer takes place in the background. The `remote_file_send()` function does not verify that the file transfer was successful. Therefore, you must ensure that the file transfer has been completed prior to using the `remote_close()` function to close the session with the remote agent.

When the file transfer is complete, an event is generated on the remote agent indicating that the file has been transferred. If the `remote_file_send()` *catalog* and *class* parameters are not supplied, an event with the following characteristics will be generated on the remote target host:

```
Event Class = RemProcess
Origin = PEM/DS,
Description = File <output_file_name>, Operation (A),
Stream Id <sendId>
```

However, by supplying an event catalog and event class to the `remote_file_send() function`, you can define the event that gets generated. You could ensure that the file transfer is complete by waiting for this event to be generated on the remote PATROL Agent using the remote_event_query() function.

Example

The following example sends a file to a remote agent and creates two events on the remote agent if the transfer is successful:

```
targetHost="<myhost>";
targetPort="<myport>";
user="<myuser>";
password="<mypassword>";
myencrypted_passwd = encrypt(password, "PEM");
source_file = "<full path to myfile>";
remote_file = "<myfile>";

print("Start remote_file_send example at ".asctime(time(),"%H:%M:%S %m-%d-%Y")."\n");
printf("TargetHost <%s> on Port <%s>\n", targetHost, targetPort);
printf("User <%s> Password <%s>\n", user, password);

# Open session to remote Agent
print("Open the channel.\n");
```

BMC Software, Inc., Confidential and Proprietary Information

```
sess=remote_open(targetHost, targetPort, "PEM", user, myencrypted_passwd);
print("errno=".errno."\n");
printf("Session #<%s> \n", sess);
#print("Sleep 5 seconds.\n");
#sleep(5);
print("Send the file.\n");

if (!sess) {
        error = "Error opening session to the Patrol Agent on the target Machine\n".
                "Please make sure the values are correct for:\n".
                "      Target Host:  " . targetHost."\n".
                "      Target Port:  " . targetPort."\n".
                "      Username:     ". user."\n".
                "      Password:     ". password."\n";
        print(error);
    }
    else {
        #+++
        # We have a session so try and send the file to remote agent
        # Specify the absolute path for the dest. file
        # Set the permissions to -rw-rw-rw- 666 Octal = 438 Decimal
        #---
        if (!remote_file_send(sess,source_file,remote_file,"STD","RemProcess","<myid>","438"))
        {
            # remote_file_send failed. Print error message\n".
            error = "Error sending file to the Patrol Agent on the target Machine\n".
                    "Please make sure the values are correct for:\n".
                    "      Target Machine: " . targetHost."\n".
                    "      Target Port:    " . targetPort."\n";
            print(error);
        }
        else
        {
           #+++
           # Wait for the file to transfer by searching for its logged event
           # This will wait for 30 seconds for the transfer to complete then
           # it gives up.
           #---
           wait_loops = 15;
           while (wait_loops > 0)
           {
               sleep(2);
               cmd_data = remote_event_query(sess,                  # Session ID
                                             "1",                   # return a maximum of 1 event
                                             "1",                   # time out in one minute
                                             "\n",                  # separate events with newline
                                             "Event ID <%{EV_ID}>", # return event id triggered
                                             "",                    # any start time
                                             "",                    # any end time
                                             "O",                   # only OPEN status
                                             "",                    # only INFORMATION type
                                             "",                    # any node
                                             "PEM/DS",              # origin
                                             remote_file,           # file name in descirption
                                             "",                    # any ID range
                                             "RemProcess",          # RemProcess event class
                                             ""                     # any severity
                                             );

                if (cmd_data != "")
                {
                    wait_loops = 0;
                }
                else
                {
                    wait_loops--;
```

```
            }
        }
        if (cmd_data == "")
        {
            error= "Error:  remote_event_query was unable to verify remote_file_send()\n".
                   "Please check the Event Manager on target:\n".
                   "      Target Machine: " . targetHost."\n".
                   "      Target Port:    " . targetPort."\n";
            print(error);
        }
        else
        {
            print("remote_file_send verified by remote_event_query: ".cmd_data);
        }
    }

    #+++
    # Close up the sesssion with the remote agent
    #---
    remote_close(sess);
}
```

The previous example creates the following events on the remote agent if the transfer is successful:

```
Event Class: RemProcess
Type:        INFORMATION
Origin:      PEM/DS
Description: File <myfile>, Operation (W), Stream Id <myid>

Event Class: Disconnect
Type:        INFORMATION
Origin:      <Source hostname>
Description: Console U:7424.3379@<source host IP address> disconnected.
```

The following example uses all the remote functions to access a remote PATROL Agent.

```
# $Id: remote_psl_exec.psl,v 1.6 1996/03/09 23:46:41 mmattini Exp $
#
# remote_psl_exec.psl
#-------------------------------------------------------------
# Description: Remote PSL execution.
#
# You need to run two agents for this example:
# From (localhost, port 1234) you want to execute a remote PSL
# script on (remotehost, port 5678).
#
# The PSL script (see your_psl_cmd below) is any PSL
# statement which return a string. It will be executed on
# (remotehost, port 5678) using PATROL console
#-------------------------------------------------------------
# Author        : Max Mattini
# Creation date : Jan 19, 1996
#
# Last check-in date: $Date: 1996/03/09 23:46:41 $
#
# Copyright (c) 1996 BMC Software, Inc.
#-------------------------------------------------------------
#
```

BMC Software, Inc., Confidential and Proprietary Information

```
# The remote agent where the PSL script is to be executed

remote_host = "remotehost";  # replace with remote host name
remote_port = "5678";        # replace with remote Agent port

# Your account details

your_username = "username";

max_time_to_wait = "1";      # maximum time (in minute) the PSL process
                             # will block waiting for the result.
your_id = "my_id";           # A local id provided by your KM to resolve
                             # any conflicts with other queries

# PSL script to be executed by <remote_host>
# It should return a string.
# Increase <max_time_to_wait> if this script
# will take a lot of time

your_psl_cmd= "get_vars(\"/\");";

#
# 0. Use an already encrypted password if you don't want to store
#    the password in the KM. It is recommended that:
#
#      - you encrypt your real password using encrypt()
#      - you store the encrypted password in a file or in the KM
#        use "pem" encryption it is safer than "des":
#        "pem" encryption is "des" encryption combined with an internal
#        encryption. Storing the result of a "pem" encryption is safe
#        since it cannot be used to log to any Unix (or other) systems.
#

your_pwd  = "becool";
your_encrypted_pwd  = encrypt(your_pwd,"pem");

#
# 1. Open the remote connection with agent (nanou,5678);
#
sess=remote_open(
   remote_host,
   remote_port,
   "pem",
   your_username,
   your_encrypted_pwd
);
print("Session number result of remote_open:".sess."\n");

print("Sending command: " . your_psl_cmd . "\n");

#
# Dispatch the remote PSL.
#

trigger_res = remote_event_trigger(
   sess,
   remote_host,              # Name of remote host
   "STANDARD",               # Name of event catalog
   "RemPsl",                 # Name of event class in StdEvents.ctg
   "INFORMATION","5",        # Type and severity
   your_psl_cmd,             # First argument to <ev_exec_name>
   your_id                   # Second argument to <ev_exec_name>
                             # This local id will uniqly identify
                             # the result of the PSL exec.
);
```

```
print("Result of trigger->".trigger_res."\n");

query_res=remote_event_query(
   sess,
   "1",                      # Finish query when 1 event is found
   max_time_to_wait,         # Time in min. PSL process will block
   "",                       # N/A (event separator)
   "%{EV_ARG1}\n",           # Format to print result of query
   "","",                    # Any start/end time
   "",                       # Any Status
   "",                       # Any Event Type
   "",                       # any node
   "",                       # any origin
   your_id,                  # pattern in filter should be your local id
   "",                       # match any eventid range in filter
   "Result"                  # match only event from class 'Result'
);
print(
   "Result of remote PSL execution on (" .
   remote_host . "," . remote_port .
   "):\n" . query_res ."<------\n");
#
# Send the file to remote agent
# The remote agent will write the content of file "/tmp/MyFile" in
# "$PATROL_REMOTE/MyRemoteFile".
# Also standard event RemProcess will be sent to the remote
# agent carrying the three dynamic arguments:
#
#   - /tmp/MyFile
#   - Operation (reserved for future use)
#   - MyFileId
#
# A notification commnad can be executed at the remote agent
# to complete the processing of the file sent
#

transfer_res = remote_file_send(
   sess,                     # remote session handle
   "/tmp/MyFile",            # full source file name
   "MyRemoteFile",           # destination name in $PATROL_REMOTE
   "STD","RemProcess",       # Use standard event FileProcess
   "MyFileId"                # my id to identify this operation
);
print("Result of transfer->".transfer_res."\n");

close_res = remote_close(sess);
print("Result of close->".close_res."\n");
```

remove()

Remove a file from the file system

Format

`remove(`*`filename`*`)`

Parameters

Parameter	Definition
filename	the name and path of the file to be removed from the file system

Description

The `remove()` function removes a specified file, *filename*, from the file system. The full path of the file must be included in the *filename* parameter.

The `remove()` function returns 1 if the file was successfully removed from the file system and returns 0 if the file could not be removed from the file system.

The `remove()` function could be unsuccessful for various reasons, including insufficient file permissions or an invalid filename.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the `file()` function to verify the existence of a file, and then removes the file using the `remove()` function. If the file exists but cannot be removed, a message is returned stating that the file properties and user permissions should be checked:

```
file_name="D:\\temp_data.txt";
   if (file(file_name)) {
   print("\nFile (".file_name.") exists!\n");
        if (remove(file_name)) {
        print("\nFile (".file_name.") removed!\n");
        } else {
        print("\nFile (".file_name.") could not be removed!\n\nCheck file
properties and user permissions!");
        }
  } else {
  print("\nFile (".file_name.") does not exist!\n");
  }
```

BMC Software, Inc., Confidential and Proprietary Information

replace()

Replace a specified string in text with a specified replacement string

Format

`replace(`*`text`*`,`*`str`*`,`*`replacement_str`*`)`

Parameters

Parameter	Definition
text	original text
str	text string to be found in *text*
replacement_str	text string to replace *str* that is found in *text*

Description

Use the `replace()` function instead of PSL common string manipulation techniques. All occurrences of *str* in *text* are replaced with *replacement_str*, and the result is returned.

Example

The following example demonstrates the `replace()` function:

```
#This example replaces the string "Hello there" with Hello world
new_text = replace("Hello there","there","world");
Hello world
```

BMC Software, Inc., Confidential and Proprietary Information

response()

Display a dialog on the PATROL Console

Format

`response(`*`title, timeout, preferences,...elements...`*`)`

Parameters

Parameter	Definition
title	character string that is the title of the dialog box The *title* is the name of the dialog box that appears in the title bar of the dialog box when it is displayed. If you are updating a dynamic response() function that has already been created, the *title* is replaced with the *rid* of the dynamic response() function dialog box. See “Dynamic response() Function Dialog Boxes” on page 1-335 for more information.
timeout	the number of seconds that the agent will wait for acknowledgment of the dialog before canceling it The *timeout* parameter defines how long the response() function waits for a PATROL Console response to the dialog box. • If *timeout* > 0, the `response()` function waits the designated number of seconds for a PATROL Console response. • If *timeout* = 0, the `response()` function returns immediately without waiting for a PATROL Console response. • If *timeout* < 0 or *timeout* = `""`, the `response()` function waits infinitely for a PATROL Console response.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
preferences	list that specifies the properties of the dialog box The *preferences* are the general characteristics of the dialog box such as size, buttons, button labels, and behavior. The *preferences* string is required, but the individual preferences are all optional. Specifying the NULL string (`""`) selects all the default preferences. **Warning:** Do not use the NULL string with the list operator. See “PSL List Operator Format” on page 1-339. See “response() Function Preferences” on page 1-339 for detailed information about the response() function preferences.
elements	lists that define the fields and controls that appear on the dialog box See “response() Function Elements” on page 1-351 for detailed information about the response() function elements.

BMC Software, Inc., Confidential and Proprietary Information

Description

The response() function defines a PATROL dialog box that is displayed by a PATROL console.

The PATROL Console will display multiple `response()` function dialogs as it receives them from the PATROL Agent. There is no restriction on the number of `response()` function dialogs that the PATROL Console is permitted to display at any given time. The number of dialogs displayed is controlled solely by the number of PSL processes executing within the PATROL Agent that are making `response()` function calls.

When you create a dialog box with the response function, you must first define the preferences of the dialog box and then define the elements that are contained in the dialog box.

Preferences of the Dialog Box

The first three parameters supplied to the response() function define the general preferences of the dialog box:

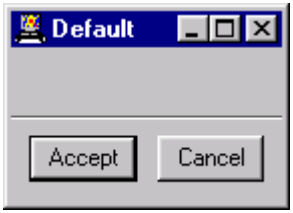
- *title*
- *timeout*
- *preferences* (See “response() Function Preferences” on page 1-339 for more information on response() function dialog box elements.)

The following response() function call produces a dialog box with the title “Default” using all the default values:

```
response("Default","","");
```

Figure 1-4 shows a response dialog produced with all default values that contains no GUI elements:

Figure 1-4 Default response() Function Dialog Box



Elements of the Dialog Box

After you define the dialog box, you define the elements that are contained in the dialog box with the element definitions. (See “response() Function Elements” on page 1-351 for more information on response() function dialog box elements.)

Note

The functionality of the response() function changes slightly from one platform to another. Considering the large number of platforms supported by PATROL, this manual cannot list all platform differences for this function. Be prepared to test this function to accommodate whatever minor irregularities exist on a specific platform.

Design Considerations of a Dialog Box

While you are working on the design of a dialog box, you should consider the following items:

- the amount of screen space
- the purpose of the dialog box
- which control elements to use
- the logical order for the control elements
- how to group or separate the control elements

BMC Software, Inc., Confidential and Proprietary Information

response() Function and the PATROL Event Manager

Calls to the `response()` function are integrated with events processed by the PATROL Event Manager. The current status of a `response()` function event, whether Open, Acknowledged, or Closed, is shown by the Event Manager display.

You can acknowledge a `response()` function event in either of the following ways:

- clicking on Acknowledge in the response() function event popup
- selecting the event and clicking on Acknowledge in the Event Manager display

In either case, assuming acknowledgment succeeds, the status change appears in the following places:

- Event Manager display
- `response()` function requests list window
- `response()` function dialog window

In the case of the `response()` function dialog window, the status is mirrored in the sensitivity of the `response()` function event popup, corresponding to the `response()` function event, rather than actually being displayed. This consistency across the three locations should apply to any status change in the `response()` function event.

A `response()` function event popup can be displayed from within the Event Manager by selecting the event and clicking on Answer.

BMC Software, Inc., Confidential and Proprietary Information

Return Values for the response() Function

The `response()` function return value is a PSL list. (Each item in a PSL list is separated by a new-line, \n, character.) The first item in the list is the *status* of the response() function, and the following items in the list are the return values of the dialog-box elements. The return value for a response() function dialog box takes the following form:

```
status\n
returnvalue01\n

. . .
returnvaluen
```

The *status* value indicates if Cancel (0) or OK (1) was selected by the user on the dialog box. Regardless of which button is selected by the user, the response() function returns a full list of values selected by the user on the dialog box.

The order of the return values is the order in which the elements are defined in the `response()` function. The following elements do not return values (not even new-line place holders):

- labels (element types 0 and 1)
- separators (element types 4 and 5)
- frames (element type 14)
- row and column compound elements (element types 15 and 16)
- icons (element type 17)

BMC Software, Inc., Confidential and Proprietary Information

response() Function Timeout Return Values

When the PATROL Console does not acknowledge a `response()` function dialog box within the timeout period, a `response()` function timeout occurs. After the timeout period, the PATROL Agent wakes the process that called the `response()` function, returns the empty string as the `response()` function return value, and sets the PSL variable *errno* to 86 (E_PSL_RESPONSE_TIMEOUT).

Note

Both Cancel and timeout have a status of 0, but Cancel returns the values of the dialog box and timeout returns the empty string.

Return Values for response() Function Elements

Table 1-11 lists the elements of the response() function that return values and describes the type of value returned by each element:

Table 1-11 Return Values for response() Function Elements (Part 1 of 3)

Return Value	Return Value Description
response() Function Status Return Value:	
status	flag identifying the button that closes the dialog box The status is always the first value in the list returned by the response() function. **Valid Values** 1 OK 0 Cancel or timeout
Element Return Values:	
Sliding Scale	*scale_value*\n integer value identifying the position of the scale pointer on the sliding scale
Radio Boxes	*radio_number*\n integer value identifying the radio button that is ON in the radio button box The radio buttons are numbered in order of their definition in the `response()` function, beginning with number 1. Each radio box returns exactly one *radio_number*.

Table 1-11 Return Values for response() Function Elements (Part 2 of 3)

Return Value	Return Value Description
Check Boxes	*cstate1 cstate2 . . . cstaten*\n space-separated row of integer values identifying the check boxes that are CHECKED(1) The *cstaten* values are listed in order of the check box definitions in the `response()` function. If no check boxes are selected, the return value is 0.
Option Menu	*option_number*\n integer value indicating the option number selected from the option menu The options are numbered in order of their definition in the response function, beginning with number 1. Each option menu returns exactly one option.
Text Fields	*text_string*\n character string value that was typed into the text field The *text_string* is a single line of text containing no new-line characters. The `response()` function return value method prevents the user from entering a new-line character in a text field request. The user can enter only one line of text.
Toggle Button	*tstate*\n binary value indicating that the toggle switch is ON (1) or OFF (0)
Single-Select Scrolled Lists	*scroll_number*\n integer value identifying the scroll list entry that is SELECTED in the single-selection scroll list The scroll list entries are numbered in order of their definition in the `response()` function, beginning with number 1. Each radio box returns exactly one *scroll_number*. If no scroll list entry is selected, the return value is 0.
Multiple-Select Scrolled List	*mscroll1 mscroll2 . . . mscrolln*\n space-separated row of integer values identifying the *n* multiple-selection scroll list entries as SELECTED (1) or UNSELECTED (0) The *mscrolln* values are listed in order of the scroll list definitions in the `response()` function. If no scroll list entries are selected, the return value is 0.

BMC Software, Inc., Confidential and Proprietary Information

Table 1-11 Return Values for response() Function Elements (Part 3 of 3)

Return Value	Return Value Description
Time Spinner	*elapsed*\n integer value that is the number of seconds that were indicated in the *hours*, *minutes*, and *seconds* windows of the spinner button The *elapsed* value is determined using the following conversion: *elapsed* = 3600(*hours*) + 60(*minutes*) + *seconds* **Valid Values** 0–86,399
popup	*pop_state*\n integer value identifying the state of the popup when it was closed The return values of the elements defined in the popup dialog box are returned with the return values of the initial response() function dialog box. The element return values follow the *pop_state* value in the order they are defined in the popup. **Valid Values** 0 popup displayed at least once and was most recently closed using **Cancel** 1 popup displayed at least once and was most recently closed using **OK** 2 popup was not displayed (initial value) 3 popup was displayed, but the parent window was closed while the popup was still open (neither **OK** nor **Cancel** was selected) A value of 0 or 1 gives no indication whether the popup was open when its parent was closed. For example, the user could display the popup, click **Cancel**, display it again, change the content, and close the main window with the popup still open. In this case, *pop_state* = 0 because the popup was last closed using the **Cancel** button.
Clicker	*value*\n integer value identifying the value on the clicker

BMC Software, Inc., Confidential and Proprietary Information

Dynamic response() Function Dialog Boxes

The dynamic `response()` function is an extension of the standard `response()` function that allows dynamic updates to the dialog box that is created on the PATROL Console.

The dynamic `response()` function is implemented using the `D` preference of the response() function. (See “response() Function Preferences” on page 1-339.) When `D=1`, the `response()` function creates a dynamic dialog box and returns the *rid* for it.

The *rid* identifier is used to perform the following tasks:

- `response()` function calls to update the dialog with new data
- `response_get_value()` calls to retrieve the status of the dialog box elements

Specifying a Dynamic response() Function Dialog Box

To create a dynamic `response()` function dialog box, call the `response()` function with the `D=1` preference. The return value from the call is the *rid*:

```
dialogID = response("Title","","D=1",
     ...elements...);
```

Subsequent calls to update the dialog box with new values use the *rid* in place of the dialog box title:

```
success = response(dialogID,"","D=1",
     ...elements...);
```

Note

Specifying the `D` preference in subsequent `response()` function calls to the dynamic dialog is optional. The `response()` function ignores the `D` preference in calls that specify *rid* instead of a *title* string.

BMC Software, Inc., Confidential and Proprietary Information

If the subsequent `response()` function is successful, it returns 1 and sets the PSL variable *errno* = 0 (E_PSL_NO_ERROR). If the *rid* is invalid or no longer exists, the subsequent `response()` function returns 0 but does not set the PSL variable *errno* and does not produce a message.

If the *rid* is so badly defined in a subsequent `response()` function that it is not recognizable as an *rid*, the `response()` function will assume it is a title and will create a dialog box with that title.

Updating a Dynamic response() Function Dialog Box

Once a dynamic dialog box has been created and the rid returned, subsequent response(*rid*,...) function calls have the following features:

- Each returns 1 if successful or 0 otherwise.
- All updates to the elements of the dynamic dialog box are immediately visible to the user.
- All updates overwrite any changes the user has made to the dynamic dialog box. The updated dynamic dialog box reflects the element definitions in the `response()` function call.
- If an element on the dynamic dialog box was modified by a user, and a `response()` function call to the dialog box did not modify the element, then the user modification will remain unchanged after the `response()` function is processed.

Along with these features, subsequent `response`(*rid*,...) function calls to a dynamic dialog box have the following restrictions:

- You must match the arguments in subsequent `response()` function calls with the arguments in the original dynamic response function call. That is, if you want to update the nth element from the original `response()` function definition, it must be the nth element in the subsequent `response`(*rid*,...) call.

BMC Software, Inc., Confidential and Proprietary Information

- The `response()` function will not allow you to replace one element type with another. The `response()` function will ignore any and all element definitions in a subsequent `response()` function call that do not match the original call.

- The `response()` function cannot be used to change the structure of a dynamic dialog box, the number of elements in the dialog box, or their order. The `response()` function can only change the values of certain elements in the dialog box.

- The `response()` function cannot be used to change the preferences of a dynamic dialog box such as the background color or the label of the OK or Cancel button.

- The `response()` function will accept empty strings (`""`) as place holders and to represent element definitions that you do not wish to modify. (Use the empty strings: repeatedly passing the original element definitions in subsequent `response()` function calls is inefficient because the `response()` function will interpret and replace the value each time even though it is unnecessary.)

Table 1-12 lists those `response()` function elements that can and cannot be updated with new values using a subsequent `response()` function call. Elements can be updated in either a dialog box or its popups. (The popup element cannot be updated, but the elements defined within it can be updated.

Table 1-12 Elements of a Dynamic response() Function

Can Update:	Cannot Update:
label (types 0, 1) sliding scale (types 2, 3) radio box (types 6, 7) check box (types 8, 9) text field (types 11, 12) toggle button (type 13) scrolled list (types 17, 18, 22, 23)	separators (types 4, 5) icon (type 19) popup (types 20, 21) frame (type 14) column (type 16) row (type 15) option menu (type 10) time spinner (type 24)

BMC Software, Inc., Confidential and Proprietary Information

Getting the Element Values of a Dynamic response() Function

The return values of the elements for a dynamic response() function dialog box are accessed using the response_get_value() function.

The `response_get_value()` function returns the most recent return value for the `response()` function dialog box identified its *rid*. The `response_get_value()` function can get the values in one of the following ways:

- block the current PSL process and wait for a return value from the response() function dialog box
- poll a queue of return values and retrieve the most recent return value for response() dialog box identified by *rid*

For more see “response_get_value()” on page 1-401.

response() Function Preferences

The preferences parameter of the response() function is a list of preferences that control how the response() function dialog box will look and behave.

Although the individual preference parameters are optional, the preferences string is required. Specify the NULL string `" "` to choose all the preference parameter defaults.

Preferences Format

You can specify the preferences using one of three formats:

- PSL list operator format
- PSL list internal representation (new-line separated)
- PSL character string format (comma separated)

PSL List Operator Format

The PSL list operator format uses the list operator to create a list of preferences. See the *PATROL Script Language Reference Manual Volume 1—PSL Essentials* for information about string operators. The PSL list operator format is less efficient than either the PSL list internal representation or the character string representation.

Warning

The list operator ignores the NULL string (`""`). The `["a", "", "b"]` list is equal to `"a\nb"`, but not `"a\n\nb"`. Use the new-line character to separate the members of a list if the list contains the NULL string.

The following example shows the response() function preferences in PSL list operator format:

```
["h=height","w=width","b=color","H=id  filename",
"e=echo","o=ok_label","c=cancel_label","N=notify",
"A=ack_required","B=broadcast","D=dynamic",
"K=end_dialog"]
```

BMC Software, Inc., Confidential and Proprietary Information

PSL List Internal Representation (New-Line Separated)

The PSL list internal representation uses the internal PSL representation of a list to define the response() function preferences. A PSL list is a character string enclosed in quotes with each item in the list separated by a new-line, \n, character. The following example shows the response() function preferences in PSL list internal representation format:

```
"h=height\nw=width\nb=color\nH=id  filename\ne=echo\n
o=ok_label\nc=cancel_label\nN=notify\nA=ack_required\n
B=broadcast\nD=dynamic\nK=end_dialog"
```

Character String Format (Comma Separated)

The PSL character string format is similar to the PSL list internal representation, but instead of each item in the list being separated by a new-line, \n, character, the character string format uses a comma. The following example shows the response() function preferences in PSL character string format:

```
"h=height,w=width,b=color,H=id  filename,e=echo,
o=ok_label,c=cancel_label,N=notify,A=ack_required,
B=broadcast,D=dynamic,K=end_dialog"
```

Note

When you are using the character string format, make sure that each item of the preference string starts with the character that designates the preference immediately following the comma with no spaces. You cannot have spaces at the beginning of a preference designation.

Preferences Parameters

Table 1-13 describes the response() function preferences parameters.

Table 1-13 response() Function Preferences (Part 1 of 9)

Parameter	Description
h=*height* and w=*width*	integer height and width of the dialog box in pixels **Valid Values** • *value* > 0 specifies the height or width of the dialog in pixels • *value* ≤ 0 specifies a dynamic height or width selected by the window manager **Default** dynamic height and width selected by the window manager The *height* and *width* parameters do not apply to PATROL Consoles running on Microsoft Windows. When a PATROL Console on Microsoft Windows displays a response() function dialog box, the dialog box is dynamically sized. If you specify a height and width on platforms that support the height and width values, scroll bars will appear if the dialog is larger than the window size.
b=*color*	character string that specifies the name of the dialog background color Do not enclose the color character string in double quotation marks. The *color* parameter maps the Motif internals of the PATROL Console machine. If Motif cannot resolve *color*, it substitutes the color that the PATROL Console uses as a background in its dialogs. **Default** colors used by PATROL

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 2 of 9)

Parameter	Description
H=*ID filename*	text string that enables a Help button on the `response()` function dialog box and specifies the file name and topic ID that the PATROL Console should display when the Help button is pressed The `H` specification must consist of a numeric *ID* and a string *filename*. The *ID* and *filename* must be separated by a delimiter character: space, colon, or minus sign. **Do not** use commas, tab characters, or new-line characters because the `response()` function will interpret these characters as field delimiters. The *filename* is either the Windows Help file for the Windows console, or the **.hlp** file that contains the Help ID mapping for the Unix console. The file name can be specified with or without the **.hlp** file extension. If the extension is omitted, PATROL adds the **.hlp** extension when the help call is made. **Default** If the `H` preference is not specified, the **Help** button does not appear on the dialog box.

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 3 of 9)

Parameter	Description
o=accept_*label* o=@*accept_label*	character string that replaces “Accept” on the Accept button for the dialog and any popups activated from the dialog Placing `@` in front of the button name disables the button from closing the dialog box when it is clicked. If the Accept button is disabled from closing the dialog box, the dialog box returns a value to the PATROL Agent and remains open for further user updates when the Accept button is clicked. This feature is generally used with dynamic response() function dialog boxes. (See “Dynamic response() Function Dialog Boxes” on page 1-335 and “response_get_value()” on page 1-401 for more information.) If you specify the `o` preference with no label (`o=`), the `response()` function does not display the Accept button on the dialog box. If you hide both the Accept and Cancel buttons, you must close the dialog box from PSL using the `K` preference. *Do not* enclose the *accept_label* character string in double quotation marks, and do not include commas, tab characters, or new-line characters in the *accept_label* character string because these characters are interpreted as field delimiters. **Default** Accept **Examples** • `o=@` disables the button labeled `Accept` from closing the dialog box • `o=Done` enables the button labeled `Done` to close the dialog box • `o=@Done` disables the button labeled `Done` from closing the dialog box • `o=` removes the button that would be labeled `Accept` from the dialog box

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 4 of 9)

Parameter	Description
c=*cancel_label* c=@*cancel_label*	character string that replaces “Cancel” on the Cancel button for the dialog and for any popups activated from the dialog Placing `@` in front of the button name disables the button from closing the dialog box when it is clicked. If the Cancel button is disabled from closing the dialog box, the dialog box returns a value to the PATROL Agent, and when the Cancel button is clicked, remains open for further user updates. This feature is generally used with dynamic response() function dialog boxes. (See “Dynamic response() Function Dialog Boxes” on page 1-335 and “response_get_value()” on page 1-401 for more information.) If you specify the `c` preference with no label (that is, `"c="`), the `response()` function does not display the “Cancel” button on the dialog box. If you hide both the Cancel and Accept buttons, you must close the dialog box from PSL using the `K` preference. *Do not* enclose the *cancellabel* character string in double quotation marks, and do not include commas, tab characters, or new-line characters in the *cancellabel* character string because these characters are interpreted as field delimiters. **Default** Cancel **Examples** • `c=@` disables the button labeled `Cancel` from closing the dialog box • `c=Accept` enables the button labeled `Accept` to close the dialog box • `c=@Accept` disables the button labeled `Accept` from closing the dialog box • `c= removes` the button that would be labeled `Accept` from the dialog box

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 5 of 9)

Parameter	Description
N=*notify*	bit flag that specifies if the dialog is editable **Valid Values** 0 the dialog box is editable The `response()` function uses the *timeout* parameter to determine how long the PATROL Agent waits for a response to the editable dialog. 1 dialog is not editable The dialog is presented to the PATROL Console as a nonblocking notification dialog. The `response()` function returns immediately without waiting for a PATROL Console response. The PATROL Console will display the noneditable dialog box until the user closes it using the **OK** or **Cancel** button. `N=1` is equivalent to specifying *timeout* = 0 in the `response()` function. **Default** 0

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 6 of 9)

Parameter	Description
A=*ack_required*	bit flag that specifies if the dialog requires an acknowledgment **Valid Values** 0 does not require explicit acknowledgment The user does not need to acknowledge the dialog box when it is presented to a single PATROL user (`B=0`) and the process that called the response() function continues when the dialog box is closed (in any manner) or when the dialog box has reached the *timeout* value. 1 changes the text of the Accept button to Acknowledge, changes the text of the Cancel button to Dismiss, and requires explicit acknowledgment of the dialog in all cases If the dialog box is not acknowledged with the Acknowledge button, no dialog box values are returned, and the PSL process that called the response() function does not continue unless the dialog box has reached the *timeout* value. When a user clicks the Acknowledge button, the status of the Response event changes from Open to Acknowledged, and the name of the button changes from Acknowledge to Accept. Clicking Accept changes the status of the event from Acknowledged to Closed, passes the return value to the PSL script, and closes the response dialog. Acknowledging a response event is different when the response dialog appears on multiple consoles (B=1). When one user clicks Acknowledge, the name of the Acknowledge button changes to Accept on all PATROL Consoles that display the dialog. When one user clicks Accept, the dialog closes on all consoles. **Default** 0

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 7 of 9)

Parameter	Description
B=*broadcast*	bit flag that specifies who receives the response to the dialog **Valid Values** 0 the PATROL Console that started the process that executed the `response()` function 1 all PATROL Consoles connected to the PATROL Agent that processed the `response()` function **Note** If a response dialog appears on multiple consoles (as allowed by B=1), the dialog requires explicit acknowledgement regardless of the setting for A=*ack_required*. **Default** 0

BMC Software, Inc., Confidential and Proprietary Information

Table 1-13 response() Function Preferences (Part 8 of 9)

Parameter	Description
D=*dynamic*	bit flag that enables the `response()` function dialog box to be dynamically updated from a PSL script; it also enables multiple return values returned to, and processed by the PSL script before the dialog box is closed For more information on dynamic response() functions, see “Dynamic response() Function Dialog Boxes” on page 1-335 and “response_get_value()” on page 1-401. After the initial `response()` function call creates a dynamic dialog box, subsequent calls to modify the field values do not require the `D` preference. The `response()` function will ignore the `D` preference if it is present. All dynamic `response()` function calls return immediately to the caller and do not use the *timeout* parameter. If you specify the *timeout* parameter on a dynamic `response()` function call, the `response()` function ignores it. Once a dynamic `response()` function dialog box has been created, further use of the `D` option in subsequent `response()` function calls to *rid* is unnecessary. The `response()` function calls to a dynamic dialog box ignore the `D` option. Dynamic `response()` function calls ignore any specified *timeout* and immediately return an *rid* or operation status value: • If *title* and `D=1` are specified, the `response()` function returns a new and unique *rid* based on host name and ID and opens a new dynamic dialog box on the PATROL Console display. • If *rid* is specified in place of *title*, the `response()` function sets the element values included in the `response()` function call and returns a 0 or 1 status value. (The state of the `D` option is irrelevant when an *rid* is specified in place of *title*.) **Valid Values** 0 disable dynamic updating of the dialog box 1 enable dynamic updating of the dialog box **Default** 0

Table 1-13 response() Function Preferences (Part 9 of 9)

Parameter	Description
K=*end_dialog*	bit flag that enables the PSL script (that created this `response()` function dialog box) also closes it This preference can only be used against a `response()` function dialog box that was created using the `D=1` preference. When specified, the dialog box will immediately close on all PATROL Consoles and return the closing status of the dialog box elements. This preference is for dynamic dialog boxes including those for which none of the existing dialog buttons will close the dialog box. **Note** If a dialog box must be acknowledged (A=1 or B=1), a user must click the appropriate button twice to close the dialog: once to change the status of the response event and once to pass the return value to the PSL script. **Valid Values** 0 do not close the `response()` function dialog box 1 immediately close the dialog box on all consoles and return the values of the dialog box elements **Default** 0

Preferences Example

The following examples show the valid formats for specifying the response() function preferences:

PSL List Internal Representation

```
# h=500dialog box height is 500 pixels
# w=300dialog box width is 300 pixels
#color change
# H=0no help button on dialog box
# e=1echo output to a text window
# o=AcceptAccept label on dialog box OK button
# c=CancelCancel label on dialog box Cancel button
# N=0dialog box is editable
# A=0acknowledgement is not required
# B=0dialog box response is not broadcast
# D=0dynamic updating is disabled
# K=0do not close dialog box from this function call
# H=1 exthlpmap help button to topic 1 in file exthlp

"h=500\nw=300\nH=0\ne=1\no=Accept\nc=Cancel\nN=0\nA=0\nB=0\nD=0\nK=0\n
H=1 exthlp",
```

PSL List Operator Format

```
[
"h=500",# dialog box height is 500 pixels
"w=300",# dialog box width is 300 pixels
# no color change
"H=0",# no help button on dialog box
"e=1",# echo output to a text window
"o=Accept",# Accept label on dialog box OK button
"c=Cancel",# Cancel label on dialog box Cancel button
"N=0",# dialog box is editable
"A=0",# acknowledgement is not required
"B=0",# dialog box response is not broadcast
"D=0",# dynamic updating is disabled
"K=0",# do not close dialog box from this function call
"H=1 exthlp"# map help button to topic 1 in file exthlp
],
```

Character String Format

```
# h=500dialog box height is 500 pixels
# w=300dialog box width is 300pixels
#no color change
# H=0no help button on dialog box
# e=1echo output to a text window
# o=AcceptAccept label on dialog box OK button
# c=CancelCancel label on dialog box Cancel button
# N=0dialog box is editable
# A=0acknowledgement is not required
# B=0dialog box response is not broadcast
# D=0dynamic updating is disabled
# K=0do not close dialog box from this function call
# H=1 exthlpmap help button to topic 1 in file exthlp

"h=500,w=300,H=0,e=1,o=Accept,c=Cancel,N=0,A=0,B=0,D=0,K=0,
H=1 exthelp",
```

BMC Software, Inc., Confidential and Proprietary Information

response() Function Elements

The *element* parameter of the response() function is one or more lists that define the elements included in the response() function dialog box. Each element list is separated by a comma. An element is any control, such as a button or text field, that is used in a dialog box. You can include up to 100 control elements in a dialog box.

Types of Elements

Every element has a specific purpose, so it is important for you to understand how each element functions when you are designing a dialog box. The following control element types are available:

- radio boxes (horizontal and vertical)
- check boxes (horizontal and vertical)
- text fields
- separator (dummy) fields
- sliding scales (horizontal and vertical)
- option menus
- labels and centered labels
- single-select and multiple-select scrolled lists
- icons
- toggle buttons
- row alignment compound element
- column alignment compound element
- frame enclosure compound element
- popup compound element
- time spinner button

Control elements can be classified as either simple control elements or compound control elements.

BMC Software, Inc., Confidential and Proprietary Information

Simple Elements

Simple control elements are used for entering data, or as simple design elements such as icons, text labels, or separators. Table 1-14 lists the simple control elements:

Table 1-14 Simple Control Elements

Control Element	Page Number
text fields	1-357
radio buttons	1-359
check boxes	1-361
toggle buttons	1-364
single-selection lists	1-366
multiple-selection lists	1-366
option menus	1-369
clickers	1-371
time spinners	1-373
sliding scales	1-375
labels	1-377
icons	1-378
separators	1-380

BMC Software, Inc., Confidential and Proprietary Information

Compound Elements

Compound control elements are containers for other elements and can contain simple elements or other compound elements. The `response()` function default action is to build a dialog box and place all specified GUI elements in a vertical column in argument order. The `response()` function syntax provides compound GUI elements to provide flexibility in constructing dialogs with nested GUI elements. The following compound GUI elements are available:

- rows
- columns
- frames
- popup dialog boxes

By default, every definition that follows a compound-element definition is part of the compound element. The compound-element definition includes a value that specifies the number of subsequent element definitions that belong to the compound element. The response() function reads the value and processes the correct number of elements into the compound element. Any remaining element definitions are processed for the parent of the compound element. Table 1-15 lists the compound elements:

Table 1-15 Compound Control Elements

Control Element	Page Number
frames	1-382
rows	1-386
columns	1-386
popup dialog boxes	1-387

Symbolic Names of Elements

Each response() function element is identified by its type. The type is either the symbolic name (a PSL constant) of the control or the numeric identifier of the control.

When you define a response() function element you can use either the number or the symbolic name to describe the graphical elements you want to display. Whenever a built-in constant is used in a PSL script, its value is substituted by the compiler with the assigned numeric value. For example, consider the following function call:

```
response("Some title"," "," ",[R_CHECK_HORIZ,...]);
```

When the PSL compiler processes this function call, the numeric value 8 is substituted for the PSL constant R_CHECK_HORIZ.

Table 1-16, “response() Function Elements,” on page 1-356 lists the numeric value, symbolic name, description, and reference page for each control that is supported by the response() function.

Nested Compound Elements

The `response()` function permits nesting of compound elements to a maximum of 50 levels with no limit on the number of elements permitted within a level.

When one compound element is nested within another, the number of elements specified for the outer does not include those specified for the inner. The response() function processes nested compound elements in the following order:

- reads and processes elements in the outer compound element until it encounters the definition of the inner
- freezes the count for the outer compound element after counting the inner definition and begins a new count for the inner
- processes elements for the inner compound until it achieves the correct number of elements
- resumes processing elements for the outer compound element

BMC Software, Inc., Confidential and Proprietary Information

Elements Format

You can specify each response() function element list using one of following formats.

- PSL list operator format
- PSL list internal representation (new-line separated)

PSL List Operator Format

The PSL list operator format uses the list operator to create a list of values that defines the element. (See the *PATROL Script Language Reference Manual Volume 1—PSL Essentials* for information about string operators.) The following example shows how an element is defined using the PSL list operator format:

```
[type,data1,data2, . . . ,datan]
```

PSL List Internal Representation (New-Line Separated)

The PSL list internal representation uses the internal PSL representation of a list to define the response() function elements. A PSL list is a character string enclosed in quotes with each item in the list separated by a new-line, \n, character. The following example shows how an element is defined using the PSL list internal representation format:

```
"type\ndata1\ndata2\n . . . \ndatan"
```

Note

If you are using the PSL list internal representation, and you are using the symbolic name of the PSL response() function element, you must not enclose the symbolic name in quotation marks. Use the following syntax for symbolic names used with the PSL list internal representation:

```
SYMBOLIC_CONSTANT."\ndata1\ndata2\n . . . \ndatan"
```

See “Symbolic Names of Elements” on page 1-354 for more information on symbolic names.

BMC Software, Inc., Confidential and Proprietary Information

Defining response() Function Elements

This section describes the elements listed in Table 1-16:

Table 1-16 response() Function Elements

No	Symbolic Name	Description	Page
0	R_LABEL	label (left-justified)	1-377
1	R_LABEL_CENTER	label (centered)	1-377
2	R_SCALE_HORIZ	sliding scale (horizontal)	1-375
3	R_SCALE_VERT	sliding scale (vertical)	1-375
4	R_SEP_HORIZ	separator (horizontal)	1-380
5	R_SEP_VERT	separator (vertical)	1-380
6	R_RADIO_VERT	radio buttons (vertical)	1-359
7	R_RADIO_HORIZ	radio buttons (horizontal)	1-359
8	R_CHECK_HORIZ	check box (horizontal)	1-361
9	R_CHECK_VERT	check box (vertical)	1-361
10	R_MENU	option menu	1-369
11	R_TEXT_FIELD	text field (without label)	1-357
12	R_TEXT_FIELD_LABEL	text field (with label)	1-357
13	R_TOGGLE	toggle button	1-364
14	R_FRAME	frame compound element	1-382
15	R_ROW	row compound element	1-386
16	R_COLUMN	column compound element	1-384
17	R_LIST_SINGLE	scrolled list (single-select, default)	1-366
18	R_LIST_MULTIPLE	scrolled list (multiple-select, defaults)	1-366
19	R_ICON	icon	1-378
20	R_POPUP_SCROLLED	popup (scrolled)	1-387
21	R_POPUP	popup (nonscrolled)	1-387
22	R_LIST_SINGLE_ND	scrolled list (single-select, no default)	1-366
23	R_LIST_MULTIPLE_ND	scrolled list (multiple-select, no defaults)	1-366
24	R_SPINNER	time spinner button	1-373
25	R_CLICKER	clicker widget	1-371

BMC Software, Inc., Confidential and Proprietary Information

Text Fields

Use text fields to allow a user to type free-form text. You can leave the text field empty for the user to fill in, or you can include default text. Figure 1-5 shows a typical text field:

Figure 1-5 Text Field Example

Element Format

[*type*,*label*,*default_text*,e=*text_echo*]

Parameter	Definition
type	integer value that identifies the element as a text field **Valid Values** 11 (R_TEXT_FIELD) text field without label 12 (R_TEXT_FIELD_LABEL) text field with label
label	character string label for the text box (not valid if *type*=11) The *label* character string must be enclosed in double quotation marks. The *label* may be specified as multiple lines of text. **Note:** omit *label* for *type*=11.
default_text	character string that is the default text within the text box If you are using the e=0 option to blank out the entered text to have a blank entry field, use the new-line character (\n) as the default text.
e=text_echo	character string that specifies whether text typed into the text box is displayed You can use the echo mode to blank out text entry for text fields that contain sensitive information, such as passwords. **Valid Values** 1 Display entered text 0 Blank out entered text **Default** 1

BMC Software, Inc., Confidential and Proprietary Information

Return Value

The response() function return value method prevents the user from entering a new-line character in a text field request. The user can only enter one line of text. The text field element returns a single line of text terminated with a new-line character and containing no internal new-line characters:

text_string`\n`

Example

The following are examples of the text field definition using the PSL list internal representation and PSL list operator format. The definition creates user name and password boxes. The password box blanks entered text.

```
[R_TEXT_FIELD_LABEL,”Username:”,”\n”,”e=1”]#text field (echo text)
[R_TEXT_FIELD_LABEL,”Password:”,”\n”,”e=0”]#text field (blank text)
```

The following examples show what happens when you forget to omit the label when *type*=11. In the first line, "LABEL" appears in the text field instead of the default text, and the default text is ignored. In the second line, the default text appears correctly in the text field.

```
[R_TEXT_FIELD, "LABEL", "Default Text"]
[R_TEXT_FIELD, "Default Text"]
```

BMC Software, Inc., Confidential and Proprietary Information

Radio Buttons

Use radio buttons to indicate a list of items that are mutually exclusive. Radio buttons require that exactly one of the radio buttons be ON and all the others be OFF. If you do not specify the initial state of the radio buttons, the `response()` function sets the first radio button ON and all the others to OFF by default. If you have numerous items in a list, and the dialog box space is limited, consider using an option menu. Figure 1-6 shows a typical radio button:

Figure 1-6 Radio Button Example

Element Format

[*type*,[*setstring*],*radion*[=*rstaten*]]

Parameter	Definition
type	integer value that identifies the element as a set of radio buttons **Valid Values** 6 (R_RADIO_VERT) vertical radio box 7 (R_RADIO_HORIZ) horizontal radio box

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
setstring	a character string in the form @*n* where *n* specifies the button that defaults to ON (The other radio buttons default to OFF.) The preceding @ character is required to differentiate *setstring* from a radio button definition. There must be no space or delimiter between the @ and the radio box number. **Example** @1 sets the first radion button that is defined as the default @3 sets the third radio button that is defined as the default
radio_n= [state_n]	*radio_n* is a character string label for the nth radio button If *setstring* is not specified for the radio button control, *radion* should be followed by =*state_n* to set the default state of the radio button. **Valid Values (state_n)** 0 OFF 1 ON Only one radio button in a radio button box is permitted to have state 1; all others must have state 0. When *setstring* is specified, the `response()` function will treat the following radio button definitions as text, so if you specify *setstring* and =*state_n*, =*state_n* appears as part of the radio button label.

Return Value

The radio buttons are numbered in order of their definition in the `response()` function, beginning with number 1. The return value is an integer value that indicates the number of the radio button that is ON.

radio_number\n

Example

In both of the following examples of the radio button definition, the second radio button, Windows, is checked by default:

```
[R_RADIO_VERT,"Unix=0","Windows NT=1","OS/2=0"],
[R_RADIO_VERT,”@2”,"Unix","Windows NT","OS/2"],
```

Check Boxes

Use check boxes when the user may select one or more items from a list of items that are not mutually exclusive or when the user is not required to select any items from the list. You may specify one or more default check boxes, but you are not required to specify any defaults. If you do not specify the initial state of the check boxes, the `response()` function sets all boxes UNCHECKED.

The check box field definition permits a maximum of 31 *label* definitions. If you require more than 31 definitions in the check box, you should create a multiple-select scrolled list. See “Lists” on page 1-366 for more information on scrolled lists.

The handling of check box return values can be aided by the PSL set functions, especially the `subset()` function, which can be used to determine whether a given check box number is set in the return value. Figure 1-7 shows a typical check box:

Figure 1-7 Check Box Example

Element Format

[*type*, [*setstring*], *checkn*[=*cstaten*]]

Parameter	Definition
type	integer value that identifies the element as a set of check boxes **Valid Values** 8 (R_CHECK_HORIZ) horizontal check box 9 (R_CHECK_VERT) vertical check box
setstring	a character string in the form `@1 b, ...` n where each n is an integer that specifies the position of a check box in the sequence of check box definitions *check1*...*checkn* that is to default to CHECKED All check box numbers not included in *setstring* default to UNCHECKED. The preceding `@` character is required to differentiate *setstring* from a button definition. There must be no space or delimiter between the `@` character and the first integer. Subsequent integers may be preceded by a single space, comma, or tab character. When *setstring* is specified, the `response()` function ignores any default settings that appear in the check box definitions. Specifying *setstring* = `@0` sets all check boxes UNCHECKED and ignores any settings specified in the button definitions.
checkn=cstaten	*checkn* is a character string label for the nth check box and may be specified as multiple lines of text *Do not* enclose the *checkn* character string in double quotation marks. *cstaten* is the binary state of the check box. **Valid Values** 0 UNCHECKED 1 CHECKED

BMC Software, Inc., Confidential and Proprietary Information

Return Value

The check boxes return a space-separated row of integer values identifying the check boxes that are CHECKED(1). The *cstaten* values are listed in order of the check box definitions in the `response()` function. If no check boxes are selected, the return value is 0.

`cstate1 cstate2 ... cstaten``\n`

Example

The following are examples of a check box definition that creates a vertical list of three check boxes titled Yes, No, and Maybe. The Maybe check box is checked, and the others are unchecked.

```
[R_CHECK_VERT,"Yes=0","No=1","Maybe=1"]
```

or

```
[9,"@2 3","Yes","No","Maybe"]
```

BMC Software, Inc., Confidential and Proprietary Information

Toggle Buttons

Use a toggle button when you have one option that may or may not be selected by a user. A toggle button is a single check box. Figure 1-8 shows a typical toggle button:

Figure 1-8 Toggle Button Example

Element Format

[*type*, *toggle*, [*tstate*]]

Parameter	Definition
type	integer value identifying the element as a toggle button **Valid Value** 13 (R_TOGGLE)
toggle	a character string that is the label for the toggle button
tstate	initial state of the toggle button **Valid Values** 0 OFF 1 ON **Default** OFF

Return Value

The toggle button returns a binary value indicating that the toggle switch is ON (1) or OFF (0).

tstate\n

Example

The following is an example of the toggle button definition using the PSL list internal representation and PSL list operator format.

```
[13,"Automatic Return in 1 Hour",1], #toggle button element
```

BMC Software, Inc., Confidential and Proprietary Information

Lists

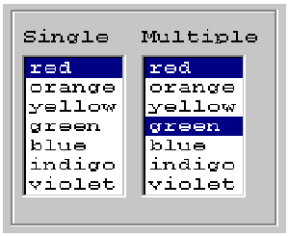
Use a list when you have a listing of items from which the user can select. You can set up a list to be a single-choice selection list or a multiple-choice selection list:

- Use a single-choice selection list when you want the user to select one option only.
- Use a multiple-choice selection list when you want the user to select more than one option.

You can set one default in a single-selection list, one or more defaults in a multiple-selection list, or no defaults at all. You cannot have more than 200 list items.

The handling of multiple-selection, scrolled-list return values can be aided by the PSL set functions, especially the `subset()` function, which can be used to determine whether a given list-element number is set in the return value. Figure 1-9 shows both a single- and multiple-selection list:

Figure 1-9 Selection List Example



Element Format

[*type*, *scrolln=sstaten*]

Parameter	Definition
type	integer value that identifies the element as a scrolled list **Valid Values** 17 (R_LIST_SINGLE) single-selection scrolled list with default 18 (R_LIST_MULTIPLE) multiple-selection scrolled list with default 22 (R_LIST_SINGLE_ND) single-selection scrolled list without default 23 (R_LIST_MULTIPLE_ND) multiple-selection scrolled list without default If *scroll-type* is 22 or 23, the `response()` function ignores the equal sign (=) and any values specified in the list definition. (This function allows you to specify display text that contains equal signs.) For scrolled lists without defaults, all entries are UNSELECTED (that is, equal to 0).
scrolln=sstaten	*scrolln* is a character string that is the label for the nth entry in the scrolled list *sstaten* is the binary state of the nth scrolled list entry **Valid Values** 0 UNSELECTED 1 SELECTED If no states are specified for the scrolled list entries, all states default to 0.

Return Value

The scrolled list entries are numbered in order of their definition in the response() function, beginning with number 1. Each scrolled list returns one or more values in a space-separated list of values (*mscrolln)* indicating the selected entries. If no scroll list entry is selected, the return value is 0.

mscroll1 . . . *mscrolln*\n

Example

The following are examples of the scrolled list definition that create four scrolled lists: single-select with default, multiple select with default, single-select without default, multiple-select without default:

BMC Software, Inc., Confidential and Proprietary Information

```
[R_LABEL,"Our Menu:"],
[R_ROW,4],
     [R_LIST_SINGLE,"Water=1","Coffee=0","Tea=0","Wine=0"],
     [R_LIST_MULTIPLE,"Okra=1","Peas=1","Corn=0","Squash=0"],
     [R_LIST_SINGLE_ND,"Beef","Hen","Pork","Fish","None"],
     [R_LIST_MULTIPLE_ND,"Cake","Pie","Fruit","Mint","Candy"],
```

BMC Software, Inc., Confidential and Proprietary Information

Option Menus

Use an option menu to indicate a list of mutually exclusive items if you do not have enough space in the dialog box for a single-selection list. Users can select the option they want by clicking the drop-down arrow or by typing the first few characters of the option's name. Figure 1-10 shows a typical option menu:

Figure 1-10 Option Menu Example

Element Format

[*type*, *label*, *default*, [*optionn*]]

Parameter	Definition
type	integer value that identifies the element as an option menu **Valid Value** 10 (R_MENU)
label	character string label for the option menu The *label* character string must be enclosed in double quotation marks.
default	integer value that identifies the default option on the option menu If no default is specified, the first option defined in the list is the default.
optionn	optional character string that identifies the text of the nth option on the option menu The *optionn* character strings must be enclosed in double quotation marks.

BMC Software, Inc., Confidential and Proprietary Information

Return Value

The options are numbered in order of their definition in the response function, beginning with number 1. Each option menu returns an integer value indicating the option number selected from the option menu.

option_number`\n`

Example

The following example creates an option menu with the label `Time` and the choices `Month`, `Day`, `Year`, `Hour`, and `Minute`, defaulting to `Day`.

```
[10,"Time:",2,"Month","Day","Year","Hour","Minute"],
```

BMC Software, Inc., Confidential and Proprietary Information

Clickers

Use a clicker if you want the user to select from a set of integer values. The user selects the appropriate value by using the up and down arrows to click through sequential values. If you are displaying the dialog box on the PATROL Console for Unix, two digits are displayed. On the PATROL Console for Windows, three digits are displayed. Figure 1-11 shows a typical clicker:

Figure 1-11 Clicker Example

Element Format

[*type*, *label*, *min*, *max*, *default-value*]

Parameter	Definition
type	value that identifies the element as a clicker **Valid Value** 25 (R_CLICKER)
label	character string that specifies the label that appears on the dialog box
min	an integer that specifies the minimum value of the clicker
max	an integer that specifies the maximum value of the clicker
default-value	an integer that specifies the default value of the clicker

BMC Software, Inc., Confidential and Proprietary Information

Return Value

The clicker returns an integer value identifying the value selected on the clicker.

value`\n`

Example

The following examples create a clicker with a label of "Select a number," a minimum value of 0, a maximum value of 100, and a default value of 50.

```
[25, "Select a number", 0, 100, 50]
```

or

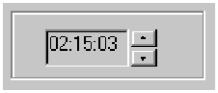
```
[R_CLICKER, "Select a number", 0, 100, 50]
```

BMC Software, Inc., Confidential and Proprietary Information

Time Spinners

A time spinner is similar to a clicker; however, a time spinner can only be used to select increments of time. The user selects the appropriate time by using the up and down arrows to click through sequential values. The hours are selected by default. The user can select the minutes or the seconds by clicking them. Figure 1-12 shows a time spinner:

Figure 1-12 Time Spinner Example



Element Format

[*type*, *seconds*]

Parameter	Definition
type	integer value identifying the element as a time spinner **Valid Values** 24 (R_SPINNER)
seconds	integer value that specifies the current time (in seconds past midnight) that will be displayed on the spinner The spinner button converts and displays the time in hh:mm:ss format. **Valid Values** 0 – 86,399

Return Value

The spinner returns the number of seconds past midnight that the user selected in the spinner.

seconds\n

Example

The following example demonstrates the spinner function by performing the following tasks:

- gets the current time
- computes the seconds past midnight to populate the spinner
- creates a response() function dialog with a spinner
- converts the return value to the corresponding time
- displays the time in a response() function dialog

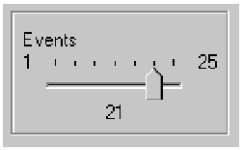
```
#get the current time and parse the return values
current_time = asctime(time(),"%S\n%M\n%H\n%Y\n%d\n%a\n%b");

seconds  = trim(nthlinef(current_time ,1),"\n");
minutes  = trim(nthlinef(current_time ,2),"\n");
hours    = trim(nthlinef(current_time ,3),"\n");
year     = trim(nthlinef(current_time ,4),"\n");
day_num  = trim(nthlinef(current_time ,5),"\n");
day_str  = trim(nthlinef(current_time ,6),"\n");
month    = trim(nthlinef(current_time ,7),"\n");

#get the number of seconds since midnight for the current time
current_seconds = 3600 * hours + 60 * minutes + seconds;

#display a dialog with the current time value in the spinner
spinner_result = response("Spinner Example","","",
   [R_LABEL,"Select a time:"],[R_SPINNER,current_seconds]);

#construct a string of the current date to find the seconds
date_string=day_str." ".day_num." ".month." ".year." 00:00:00";

#get the number of seconds until midnight
base=convert_date(date_string);

#get the spinner result
spinner_result = trim(nthlinef(spinner_result,2),"\n");
#get a character string of the selected time
spinner_result = asctime(spinner_result+base);

#display the selected time
response("Spinner Example","","",
   [R_LABEL,"Time selected:"],[R_LABEL,spinner_result]);
```

BMC Software, Inc., Confidential and Proprietary Information

Sliding Scales

Use a sliding scale to allow the user to graphically select a value within a range by moving the slider arm to the correct position. All sliding scale values must be integers. If you are displaying the dialog box on the PATROL Console for Windows, two digits are displayed. On the PATROL Console for Unix, only the default value is displayed. When you create a sliding scale, the slider arm points to the relative location of the default value on the slider shaft, and the numeric default value appears centered on the slider shaft, as shown in Figure 1-13.

Figure 1-13 Sliding Scale Example



The response() function performs the following consistency checks on the sliding scale *minimum*, *maximum*, and *initial* values:

- If *minimum* > *maximum*, set *maximum* = *minimum* + 100
- If *initial* < *minimum* or *initial* > *maximum*, set *initial* = (*minimum* + *maximum*) / 2

Element Format

[*type*, *scale-label*, *minimum*, *maximum*, *initial*]

Parameter	Definition
type	integer value that identifies the element as a sliding scale definition **Valid Values** 2 (R_SCALE_HORIZ) horizontal sliding scale 3 (R_SCALE_VERT) vertical sliding scale
scale-label	character string label for the sliding scale The *scale-label* character string must be enclosed in double quotation marks.

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
minimum	numeric value specifying the lower limit of the scale **Default** 0
maximum	numeric value indicating the upper limit of the scale **Default** 100
initial	numeric value indicating the initial position of the scale pointer on the sliding scale **Default** midway between *minimum* and *maximum*

Return Value

The sliding scale returns the value selected by the user.

value\n

Example

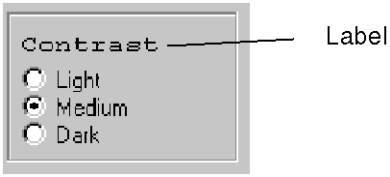
The following are examples of the sliding scale definition. The definition pops up a set of sliding bar “Time Machine” controls defaulting to the current date and time.

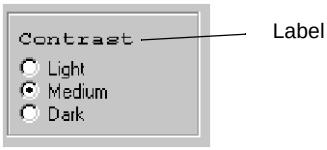
```
[21,1,"Time Machine","Time Controls"],     #popup dialog box
     [16,2],                               #column compound element
          [14,1],                          #frame compound element
               [16,4],                     #column compound element
                  [1,"Clock Controls"],    #label element
                  [2,"seconds",0,59,""],   #slide bar element
                  [2,"minutes",0,59,""],   #slide bar element
                  [2,"hours",0,23,""],     #slide bar element
          [14,1],#frame compound element:
               [16,4],                     #column compound element
                  [1,"Calendar Controls"], #label element
                  [2,"day",1,31,""],       #slide bar element
                  [2,"month",1,12,""],     #slide bar element
                  [2,"year",""],           #slide bar element
```

Labels

Use labels to provide users with information about a control element or group of control elements. Labels help users understand the dialog box. Figure 1-14 shows a typical label:

Figure 1-14 Label Example





Element Format

[*type*, *label*]

Parameter	Definition
type	integer value that identifies the element as a label **Valid Value** 0 (R_LABEL) left-justified label 1 (R_LABEL_CENTER) centered label
label	character string that specifies the label that appears on the dialog box The *label* character strings must be enclosed in double quotation marks. The *label* may be specified as multiple lines of text by using the new-line character (\n).

Return Value

The label element does not have a return value.

Example

The following definition creates a label for a BMC Software logo.

```
[R_LABEL_CENTER,"The BMC Software logo follows:"],
[R_ICON,"$PATROL_HOME/lib/images/BMC.xbm"],
```

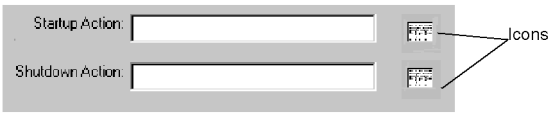
Icons

Use icons to display graphics, such as logos or icons, in a dialog box.

The creation of icons on the PATROL Console can sometimes consume significant processing time. Unless an icon is already cached internally, it takes time to load the image from disk.

The second time a given icon is requested, the performance penalty will disappear if the icon is still in cache. The performance penalty may also disappear if the icon already appears elsewhere within the console GUI. However, the caching effect leading to better performance may only occur if you have actually viewed the window in which the icon appears. Figure 1-15 shows a typical icon:

Figure 1-15 Icon Example



Element Format

[*type*, *icon-filename*]

Parameter	Definition
type	integer value identifying the element as an icon **Valid Value** 19 (R_ICON)
icon-filename	character string identifying the file containing the icon image The *icon-filename* character string must be enclosed in double quotation marks. If the file extension is omitted, the response function assumes an extension of **.gif**.

Return Value

The icon element does not have a return value.

Example

The following example displays the BMC Software logo (**BMC.xbm**)from the directory ***$PATROL_HOME*/lib/images** in the dialog box.

```
[R_ICON,"BMC.xbm"],
```

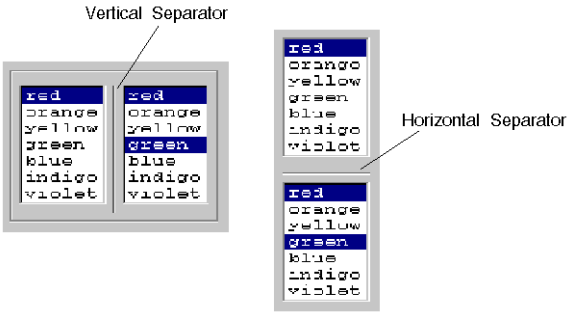
Separators

Use separators to help users distinguish between different control elements or groups of control elements. You can use vertical separators or horizontal separators. Vertical separators must be put inside rows; horizontal separators must be put inside columns.

The separator places a line on the dialog box and allows you to format and group the other fields you place on the dialog box.

Figure 1-16 shows vertical and horizontal separators:

Figure 1-16 Separator Example



Element Format

[*type*]

Parameter	Definition
type	integer value that identifies the element as a separator **Valid Values** 4 (R_SEP_HORIZ) horizontal separator 5 (R_SEP_VERT) vertical separator

Return Value

The separator element does not have a return value.

Example

The following example separates two columns of radio buttons with a vertical separator.

```
[15,3],                                                 #   row compound element:
    [6,"Unix=1","Windows NT=0","OS/2=0"],                #       radio button
    [5],                                                 #       vertical separator
    [6,"Binary=0","Text=0","Autoselect=1"],              #       radio button
```

Frames

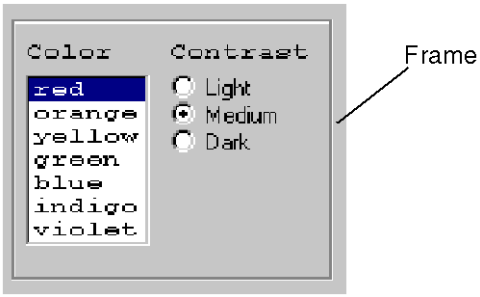
Use frames to group related control elements together. Frames can help to set related control elements apart from other control elements or from groups of control elements in the dialog box.

The element definitions that are part of a frame compound definition are placed in an enclosed bordered region of the dialog or popup. The frame elements are aligned in the following manner:

- If the element is specified in a column, it is positioned on the left edge of the frame that contains it.
- If the element is specified in a row, it is positioned immediately to the right of the previous element definition.

Figure 1-17 shows a typical frame:

Figure 1-17 Frame Example



Element Format

[*type*, *felements*]

Parameter	Definition
type	integer value identifying the element as a frame **Valid Value** 14 (R_FRAME)
felements	integer value specifying the number of subsequent element definitions that are included in the frame The *felements* value includes nested compound element definitions but does not include the elements that are specified for those nested compound elements. Refer to “Nested Compound Elements” on page 1-354.

Note

The `response()` function ignores the *`felements`* parameter and sets its value to 1. To define multiple elements in a frame, define a frame element and then define a row or column as the frame’s first element.

Return Value

The frame element does not have a return value.

Example

The following example creates a frame enclosing a column of three rows of radio buttons, three buttons in each row.

```
[14,1],
     [16,3],
          [7,"Yes=0","No=0","Maybe=1"],
          [7,"Don’t Know=0","Don’t Care=0","Good-bye=1"],
          [7,"Great=0","Good=0","So-So=1"],
```

BMC Software, Inc., Confidential and Proprietary Information

Columns

You can use columns to position control elements in a dialog box. Columns position control elements vertically.

All of the control elements in a response() function dialog box must be put inside rows or columns. You can put rows inside columns and columns inside rows. Rows and columns are used for design purposes only; they are invisible to the user.

The element definitions that are part of a column-compound definition are placed vertically from top to bottom in the dialog or popup. The column is aligned along the left edge of the frame that contains it, immediately beneath the previous element definition.

Unless otherwise specified, all response() function definitions are aligned in a single row. Figure 1-18 shows a typical column:

Figure 1-18 Column Example

Lists and Separator Inside a Column

Element Format

[*type*, *celements*]

Parameter	Definition
type	integer value identifying the element as a column **Valid Value** 16 (R_COLUMN)
celements	integer value specifying the number of subsequent element definitions that are included in the column The *celements* value includes nested compound-element definitions but does not include the elements that are specified for those nested compound elements. Refer to “Nested Compound Elements” on page 1-354.

Return Value

The column element does not have a return value.

Example

The following example creates a column of three rows of radio buttons, three buttons in each row.

```
[16,3],
   [7,"Yes=0","No=0","Maybe=1"],
   [7,"Don’t Know=0","Don’t Care=0","Good-bye=1"],
   [7,"Great=0","Good=0","So-So=1"],
```

BMC Software, Inc., Confidential and Proprietary Information

Rows

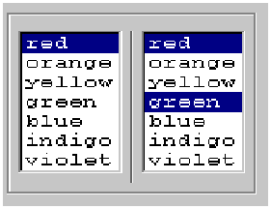
You can use rows to position control elements in a dialog box. Rows position control elements horizontally.

All of the control elements in a response() function dialog box must be put inside rows or columns. You can put rows inside of columns and columns inside of rows. Rows and columns are used for design purposes only; they are invisible to the user.

The element definitions that are part of a row-compound definition are placed horizontally from left to right across the dialog box or popup. The row is aligned beginning at the left edge of the frame that contains it, immediately beneath the previous element definition. Figure 1-19 shows a typical row:

Figure 1-19 Row Example

Lists and Separator Inside a Row



Element Format

[*type*, *relements*]

Parameter	Definition
type	integer value identifying the element as a row **Valid Values** 15 (R_ROW)
relements	integer value specifying the number of subsequent element definitions that are included in the row The *relements* value includes nested-compound element definitions but does not include the elements that are specified for those nested-compound elements. Refer to “Nested Compound Elements” on page 1-354.

Return Value

The row element does not have a return value.

Example

The following example creates a row with three elements: two columns of radio buttons and a vertical separator between them.

```
[15,3],#row compound element:
     [6,"Unix=1","Windows NT=0","OS/2=0"],        #radio button
     [5],                                         #vertical separator
     [6,"Binary=0","Text=0","Autoselect=1"],      #radio button
```

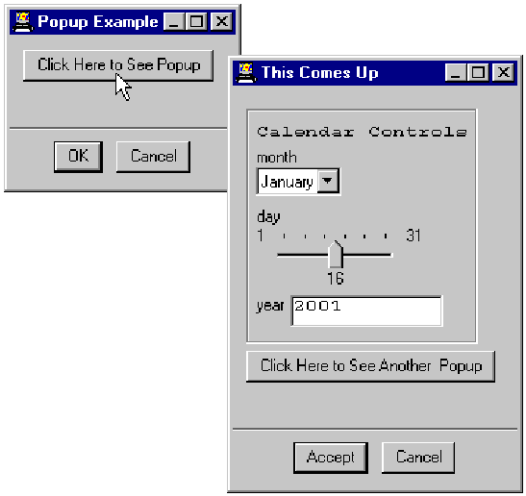
Popup Dialog Boxes

You can include a secondary or popup dialog box in a main dialog box. The secondary dialog box may contain all of the control elements that a main dialog box can contain, including its own popup dialog box.

BMC Software, Inc., Confidential and Proprietary Information

The popup is a compound GUI element that is displayed by clicking on a popup button in the parent dialog box. The element definitions included in the popup definition are displayed in the popup dialog box. The element definitions for popups can include other popups and compound elements as well as common element definitions (like buttons, lists, and check boxes). Figure 1-20 shows a typical popup:

Figure 1-20 Popup Example



Element Format

[*type*, *pelements*, *btitle*, *ptitle*, *h=pheight*, *w=pwidth*, *b=pcolor*]

Parameter	Definition
type	integer value identifying the element as a popup **Valid Values** 20 (R_POPUP_SCROLLED) scrolled popup 21 (R_POPUP) nonscrolled popup
pelements	integer value specifying the number of subsequent element definitions that are included in the popup The *pelements* value includes nested compound-element definitions but does not include the elements that are specified for those nested compound elements. Refer to “Nested Compound Elements” on page 1-354.
btitle	character string that is the label for the popup button on the parent popup or dialog box **Default** popup
ptitle	character string that is the title of the popup dialog box **Default** popup
h=pheight	integer height of the popup in pixels On Windows the *pheight* and *pwidth* values are ignored, and the popup height and width are dynamically set by Windows. **Valid Values** • *pheight* > 0 the height of the popup in pixels • *pheight* ≤ 0 a dynamic height **Default** dynamic height selected by the window manager

BMC Software, Inc., Confidential and Proprietary Information

Parameter	Definition
w=pwidth	integer width of the popup in pixels On Windows the *pheight* and *pwidth* values are ignored, and the popup height and width are dynamically set by Windows. **Valid Values** • *pwidth*> 0 the width of the popup in pixels • *pwidth* ≤ 0 a dynamic width **Default** dynamic height selected by the window manager
b=pcolor	character string that specifies the name of the popup background color *pcolor* maps the Motif internals of the PATROL Console machine. If Motif cannot resolve *pcolor*, it substitutes the color that the PATROL Console uses.

Return Value

The popup returns an integer value identifying the state of the popup when it was closed.

`pop_state``\n`

- 0–popup displayed at least once and was most recently closed using **Cancel**
- 1–popup displayed at least once and was most recently closed using **OK**
- 2–popup was not displayed (initial value)
- 3–popup was displayed, but the parent window was closed while the popup was still open (neither **OK** nor **Cancel** was selected)

The return values of the elements defined in the popup dialog box are returned with the return values of the initial response() function dialog box. The element return values follow the *pop_state* value in the order they are defined in the popup.

Example

The following example creates a dialog box with a popup. This example response() function code produces the dialog box shown in Figure 1-20 on page 1-388.

```
response("Popup Example","",["o=OK"],
   [21,2,"Click Here to See Popup","This Comes Up"],#popup 2 elements
     [14,1], #frame with 1 element
       [16,4], #column with 4 elements
         [1,"Calendar Controls"], #label
         [10,"month","January","February", #option menu
           "March","April","May","June","July", #options
           "August","September","October", #options
           "November","December"], #options
         [2,"day",1,31,16], #sliding scale
         [12,"year",2001], #text field
  [21,1,"Click Here to See Another Popup","Another Popup"],#popup
     [1,"Another Popup"] #label
);
```

Example

The following PSL script combines all the previous element examples to produce a `response()` function dialog box using PSL list internal representation and PSL list operator definitions. The script consists of

- a `response_analysis()` function that decodes and provides a text summary of the `response()` function return value
- a `main()` function that displays the response dialog and then calls the `response_analysis()` function to decode the return value

The `response()` function contains comments to assist you in recognizing the element definitions.

BMC Software, Inc., Confidential and Proprietary Information

```
function response_analysis(status) {

   # status line 1 is the check box status

   print("Check box status:\n");
   checks = nthlinef(status,1);
   i = 1;
   temp = nthargf(checks,i);
   while(temp) {
      switch (temp) {
         case 1: {print("  Yes is checked\n");}
         case 2: {print("  No is checked\n");}
         case 3: {print("  Maybe is checked\n");}
      }
      i++;
      temp = nthargf(checks,i);
   }
   print("\n");

   # status lines 2 through 4 are the framed rows of radio boxes statuses:

   print("Radio box frame status:\n");
   i = 2;
   while (i <= 4) {
      switch (i) {
         case 2: {label = ["Yes","No","Maybe"];}
         case 3: {label = ["Left","Center","Right"];}
         case 4: {label = ["More","Enough","Less"];}
      }
      radio = int(nthlinef(status,i));
      switch (radio) {
         case 1: {print("  ",trim(nthlinef(label,1),"\n"),"  is selected\n");}
         case 2: {print("  ",trim(nthlinef(label,2),"\n"),"  is selected\n");}
         case 3: {print("  ",trim(nthlinef(label,3),"\n"),"  is selected\n");}
      }
      i++;
   }
   print("\n");

   # status line 5 is the option menu status

   print("Option menu status:\n");
   option = int(nthline(status,5));
   switch (option) {
      case 1: {print("  Month is selected\n\n");}
      case 2: {print("  Day is selected\n\n");}
      case 3: {print("  Year is selected\n\n");}
      case 4: {print("  Hour is selected\n\n");}
      case 5: {print("  Minute is selected\n\n");}
   }

   # status line 6 is the popup close status

   print("popup dialog close status:\n");
   popup = int(nthline(status,6));
   switch (popup + 1) {
      case 1: {print("  popup was displayed and closed using Cancel\n\n");}
      case 2: {print("  popup was displayed and closed using Accept\n\n");}
      case 3: {print("  popup was not displayed\n\n");}
      case 4: {print("  popup was open when parent dialog closed\n\n");}
   }

   # status lines 7 through 10 are the time control values

   print("Time control settings:\n");
   control_time = trim(nthlinef(status,7),"\n");
```

BMC Software, Inc., Confidential and Proprietary Information

```
hours = int(control_time / 3600);
printf("  hours control is set to %d\n",hours);
control_time = control_time - (hours * 3600);
minutes = int(control_time / 60);
printf("  minutes control is set to %d\n",minutes);
seconds = control_time - (minutes * 60);
printf("  seconds control is set to %d\n",seconds);
month_string = [
   "January",
   "February",
   "March",
   "April",
   "May",
   "June",
   "July",
   "August",
   "September",
   "October",
   "November",
   "December"
];
printf("  month control is set to %s",nthlinef(month_string,trim(nthlinef(status,8),"\n")));
i = 9;
while (i <= 10) {
   quantity = ["day","year"];
   print("  ",trim(nthlinef(quantity,i - 8),"\n"),," control is set to ",nthlinef(status,i));
   i++;
}
print("\n");

# status line 11 is the toggle status

print("Automatic return toggle status:\n");
if (int(nthlinef(status,11)) == 1) {
   print("  Automatic return in 1 hour is set\n\n");
} else {
   print("  Automatic return in 1 hour is NOT set\n\n");
}

# status lines 12 and 13 are the vertically separated radio box statuses:

print("Radio box status:\n");
i = 12;
while (i <= 13) {
   switch (i) {
      case 12: {label = ["Unix","Windows NT","OS/2"];}
      case 13: {label = ["Binary","Text","Autoselect"];}
   }
   radio = int(nthlinef(status,i));
   switch (radio) {
      case 1: {print("  ",trim(nthlinef(label,1),"\n"),," is selected\n");}
      case 2: {print("  ",trim(nthlinef(label,2),"\n"),," is selected\n");}
      case 3: {print("  ",trim(nthlinef(label,3),"\n"),," is selected\n");}
   }
   i++;
}
print("\n");

# status line 14 is the single-select scrolling menu with default

print("Single-select scrolling menu status:\n");
option = int(nthline(status,14));
switch (option) {
   case 1: {print("  Water is selected\n\n");}
   case 2: {print("  Coffee is selected\n\n");}
   case 3: {print("  Tea is selected\n\n");}
```

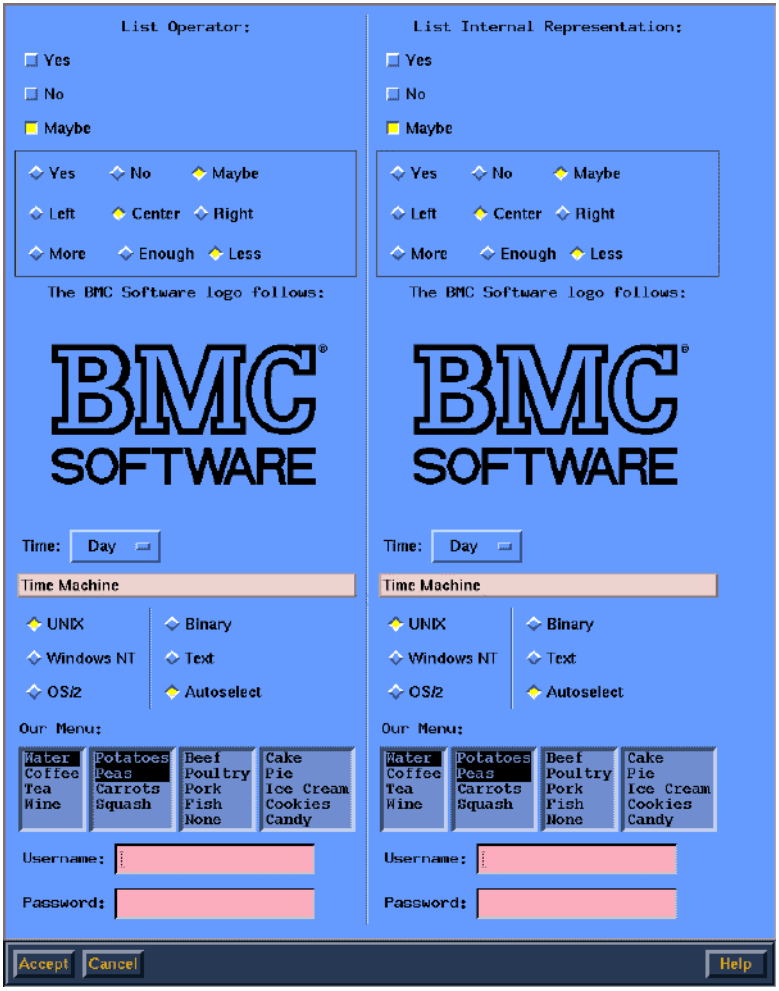
```
      case 4: {print("  Wine is selected\n\n");}
   }

   # status line 15 is the multiple-select scrolling menu with default

   print("Multiple-select scrolling menu status:\n");
   option = nthlinef(status,15);
   i = 1;
   temp = nthargf(option,i);
   while(temp) {
      switch (temp) {
         case 1: {print("  Potatoes are selected\n");}
         case 2: {print("  Peas are selected\n");}
         case 3: {print("  Carrots are selected\n");}
         case 4: {print("  Squash is selected\n");}
      }
      i++;
      temp = nthargf(option,i);
   }
   print("\n");

   # status line 16 is the single-select scrolling menu with no default

   print("Single-select scrolling menu status:\n");
   option = int(nthline(status,16));
   switch (option) {
      case 1: {print("  Beef is selected\n\n");}
      case 2: {print("  Poultry is selected\n\n");}
      case 3: {print("  Fish is selected\n\n");}
      case 4: {print("  None is selected\n\n");}
   }

   # status line 17 is the multiple-select scrolling menu with default

   print("Multiple-select scrolling menu status:\n");
   option = nthlinef(status,17);
   i = 1;
   temp = nthargf(option,i);
   while(temp) {
      switch (temp) {
         case 1: {print("  Cake is selected\n");}
         case 2: {print("  Pie is selected\n");}
         case 3: {print("  Ice Cream is selected\n");}
         case 4: {print("  Cookies are selected\n");}
         case 5: {print("  Candy is selected\n");}
      }
      i++;
      temp = nthargf(option,i);
   }
   print("\n");

   # status line 18 is the username field content:

   print("Username:\n");
   print("  ",nthline(status,18),"\n");

   # status line 19 is the password field content:

   print("Password field content:\n");
   print("  ",nthline(status,19),"\n");

   return;
}

function main() {
   psl_tm = asctime(time(),"%S\n%M\n%H\n%d\n%m\n%y\n%w\n%j");
```

```
seconds = trim(nthlinef(psl_tm,1),"\n");
minutes = trim(nthlinef(psl_tm,2),"\n");
hours = trim(nthlinef(psl_tm,3),"\n");
day = trim(nthlinef(psl_tm,4),"\n");
month = trim(nthlinef(psl_tm,5),"\n");
year = trim(nthlinef(psl_tm,6),"\n") + 1900;
current = 3600 * hours + 60 * minutes + seconds;

return_status = response(
   "Dean's response() Example",                                  # dialog box title
   -1,                                                           # infinite timeout
   ["h=910","w=750","H=0","e=1","o=Accept","N=0","A=0","B=0"],   # preferences

   # The following are the dialog box control definitions.
   # Those controls that return a value are identified by an
   # asterisk preceding the label in the comment area on the
   # right-hand side.

   [15,3],                                                       # row:
      [16,12],                                                   #   column:
         [1,"List Operator:"],                                   #     label
         [9,"Yes=0","No=0","Maybe=1"],                           #     *check box
         [14,1],                                                 #     frame:
            [16,3],                                              #       column:
               [7,"Yes=0","No=0","Maybe=1"],                     #         *radio box
               [7,"Left=0","Center=1","Right=0"],                #         *radio box
               [7,"More=0","Enough=0","Less=1"],                 #         *radio box
         [1,"The BMC Software logo follows:"],                   #     label
         [19,"$PATROL_HOME/lib/images/BMC.xbm"],                 #     icon
         [10,"Time:",2,"Month","Day","Year","Hour","Minute"],    #     *option menu
         [21,1,"Time Machine","Time Controls"],                  #     *popup:
            [16,4],                                              #       column:
               [14,1],                                           #         frame:
                  [16,2],                                        #           column:
                     [1,"Clock Controls"],                       #             label
                     [24,current],                               #             *time spinner
               [14,1],                                           #         frame:
                  [16,4],                                        #           column:
                     [1,"Calendar Controls"],                    #             label
                     [10,"month",month,"January","February",     #             *option menu
                        "March","April","May","June","July",     #
                        "August","September","October",          #
                        "November","December"],                  #
                     [2,"day",1,31,day],                         #             *slide bar
                     [12,"year",int(year)],                      #             *text field
               [13,"Automatic Return in 1 Hour",1],              #         *toggle button
               [1,"Welcome to another time!"],                   #         label
         [15,3],                                                 #     row:
            [6,"Unix=1","Windows NT=0","OS/2=0"],                #       *radio button
            [5],                                                 #       vertical separator
            [6,"Binary=0","Text=0","Autoselect=1"],              #       *radio button
         [0,"Our Menu:"],                                        #     label
         [15,4],                                                 #     row:
            [17,"Water=1","Coffee=0","Tea=0","Wine=0"],          #       *scrolled list s,d
            [18,"Potatoes=1","Peas=1","Carrots=0","Squash=0"],   #       *scrolled list m,d
            [22,"Beef","Poultry","Pork","Fish","None"],          #       *scrolled list s,~d
            [23,"Cake","Pie","Ice Cream","Cookies","Candy"],     #       *scrolled list m,~d
         [12,"Username:"," ","e=1"],                             #     *text field (echo)
         [12,"Password:"," ","e=0"],                             #     *text field (blank)
      [5],                                                       #   vertical separator
      "16\n12",                                                  #   column:
         "1\nList Internal Representation:",                     #     label
         "9\nYes=0\nNo=0\nMaybe=1",                              #     *check box
         "14\n1",                                                #     frame:
            "16\n3",                                             #       column:
```

```
                 "7\nYes=0\nNo=0\nMaybe=1",                      #         *radio box
                 "7\nLeft=0\nCenter=1\nRight=0",                 #         *radio box
                 "7\nMore=0\nEnough=0\nLess=1",                  #         *radio box
              "1\nThe BMC Software logo follows:",               #     label
              "19\n$PATROL_HOME/lib/images/BMC.xbm",             #     icon
              "10\nTime:\n2\nMonth\nDay\nYear\nHour\nMinute",    #     *option menu
              "21\n1\nTime Machine\nTime Controls",              #     *popup:
                 "16\n4",                                        #       column:
                    "14\n1",                                     #         frame:
                       "16\n2",                                  #           column:
                          "1\nClock Controls",                   #             label
                          "24\n" . current,                      #             *time spinner
                    "14\n1",                                     #         frame:
                       "16\n4",                                  #           column:
                          "1\nCalendar Controls",                #             label
                          "10\nmonth\n" . month . "\nJanuary\n" .  #           *option menu
                             "February\nMarch\nApril\nMay\nJune\n" . #
                             "July\nAugust\nSeptember\nOctober\n" .  #
                             "November\nDecember",               #
                          "2\nday\n1\n31\n" . day,               #             *slide bar
                          "12\nyear\n" . int(year),              #             *text field
                    "13\nAutomatic Return in 1 Hour\n1",         #         *toggle button
                    "1\nWelcome to another time!",               #         label
              "15\n3",                                           #     row:
                 "6\nUnix=1\nWindows NT=0\nOS/2=0",              #       *radio button
                 "5",                                            #       vertical separator
                 "6\nBinary=0\nText=0\nAutoselect=1",            #       *radio button
              "0\nOur Menu:",                                    #     label
              "15\n4",                                           #     row:
                 "17\nWater=1\nCoffee=0\nTea=0\nWine=0",         #       *scrolled list s,d
                 "18\nPotatoes=1\nPeas=1\nCarrots=0\nSquash=0",  #       *scrolled list m,d
                 "22\nBeef\nPoultry\nPork\nFish\nNone",          #       *scrolled list s,~d
                 "23\nCake\nPie\nIce Cream\nCookies\nCandy",     #       *scrolled list m,~d
              "12\nUsername:\n\ne=1",                            #     *text field (echo)
              "12\nPassword:\n\ne=0"                             #     *text field (blank)
   );

   print("Analysis of response() return value:\n");
   print("-----------------------------------\n\n");

   # status line 1 is the dialog close status

   print("Dialog close status:\n");
   if (int(nthlinef(return_status,1)) == 1) {
      print("  Dialog closed with Accept button\n\n");
   } else {
      print("  Dialog closed with Cancel button\n\n");
   }

   print("Analysis of response() dialog list operator fields:\n");
   print("-----------------------------------------------------\n\n");
   list_op_status = nthlinef(return_status,"2-20");
   response_analysis(list_op_status);

   print("\n");
   print("Analysis of response() dialog list internal representation fields:\n");
   print("-------------------------------------------------------------------\n\n");
   internal_rep_status = tail(return_status,19);
   response_analysis(internal_rep_status);

   return;
}
```

Figure 1-21 shows the `response()` function dialog box displayed by the example:

Figure 1-21 response() Function Example Dialog Box



The following is a sample output from the `response()` function example:

```
Analysis of response() return value:
------------------------------------

Dialog close status:
```

```
  Dialog closed with Accept button

Analysis of response() dialog list operator fields:
---------------------------------------------------

Check box status:
  Yes is checked

Radio box frame status:
  Yes is selected
  Left is selected
  More is selected

Option menu status:
  Month is selected

popup dialog close status:
  popup was displayed and closed using Accept

Time control settings:
  hours control is set to 5
  minutes control is set to 42
  seconds control is set to 22
  month control is set to August
  day control is set to 17
  year control is set to 1955

Automatic return toggle status:
  Automatic return in 1 hour is set

Radio box status:
  Windows NT is selected
  Binary is selected

Single-select scrolling menu status:
  Coffee is selected

Multiple-select scrolling menu status:
  Peas are selected
  Squash is selected

Single-select scrolling menu status:
  Beef is selected

Multiple-select scrolling menu status:
  Cake is selected
  Pie is selected

Username:
  dherr1

Password field content:
  pass1


Analysis of response() dialog list internal representation fields:
------------------------------------------------------------------

Check box status:
  No is checked

Radio box frame status:
  Maybe is selected
  Right is selected
  Less is selected
```

```
Option menu status:
  Year is selected

popup dialog close status:
  popup was not displayed

Time control settings:
  hours control is set to 9
  minutes control is set to 2
  seconds control is set to 22
  month control is set to June
  day control is set to 19
  year control is set to 1996

Automatic return toggle status:
  Automatic return in 1 hour is set

Radio box status:
  Unix is selected
  Autoselect is selected

Single-select scrolling menu status:
  Tea is selected

Multiple-select scrolling menu status:
  Peas are selected
  Carrots are selected

Single-select scrolling menu status:
  None is selected

Multiple-select scrolling menu status:
  Cookies are selected
  Candy is selected

Username:
  dherr2

Password field content:
  pass2
```

response_get_value()

Retrieve a return value from the elements of a dynamic `response()` function dialog box

Format

`response_get_value(`*`rid`*`,[`*`polltype`*`])`

Parameters

Parameter	Definition
rid	the identifier of the `response()` function dialog box from which it gets values The `response()` function returns an *rid* instead of an element return value the first time it is executed with the `D=1` preference specified. See “response() Function Preferences” on page 1-339 for more information on the D preference.
polltype	polling type the `response_get_value()` function uses to acquire the return value of the dialog box identified by *rid* **Valid Values** 0 block the current PSL process and wait for the dialog box to return a value 1 poll the queue of return values from `response()` function dialog boxes and return the most recent return value that is queued for *rid* **Default** 0

BMC Software, Inc., Confidential and Proprietary Information

Description

The `response_get_value()` function retrieves the most recent return value for the `response()` function dialog box identified by *rid*. Depending on the value of *polltype*, the `response_get_value()` function will either block the current PSL process and wait for a return value or will poll a queue of return values and retrieve the most recent return value for *rid*.

The `response_get_value()` function is required to get the values of the elements for a dynamic `response()` function dialog box because a dynamic response() function dialog box does not return element values when it is created, updated, or closed. The dynamic response() function only returns operation status values.

The `response_get_value()` function returns one of the following *errno* values on failure:

errno Value	Description
84	E_PSL_RESPONSE_NO_VALUE The dialog is gone and no queued values are available.
84	E_PSL_RESPONSE_NO_VALUE No data is available for a non-blocking call.
93	E_PSL_NO_SUCH_ID An invalid response ID was specified.
128	E_PSL_ALREADY_WAITING Another process is waiting on the ID.

Blocking for a response() Function Return Value

Specifying *polltype* = 0 causes the `response_get_value()` function to block the PSL process from which it is called and wait for a return value from the `response()` function dialog box identified by *rid*. There is no timeout value for the `response_get_value()` function in blocking mode, the function must receive a return value from the response() function dialog box identified by *rid*.

BMC Software, Inc., Confidential and Proprietary Information

A dynamic `response()` function dialog box returns a value each time the `o` (Accept) or `c` (Cancel) button on the dialog box is clicked or when the dialog box is closed.

Polling for a response() Function Return Value

Specifying *polltype* = 1 causes the `response_get_value()` function to query a queue that contains return values from all `response()` function dialog boxes:

- If *rid* has an invalid format, the `response_get_value()` function returns the NULL string and produces a PSL run-time error message (*errno* = 93).

- If *rid* has a valid format but the `response()` function dialog box is still active and has no return values in the queue, the `response_get_value()` function returns the NULL string and produces a PSL run-time error message (*errno* = 84).

- If *rid* has a valid format but the `response()` function dialog box is no longer active and has no more return values in the queue, the `response_get_value()` function returns the NULL string and produces a PSL run-time error message (*errno* = 84).

- If *rid* has a valid format and the `response()` function dialog box is active and has returned one or more return values to the queue, the `response_get_value()` function returns the most recent of the return values for *rid* and removes it from the queue.

BMC Software, Inc., Confidential and Proprietary Information

rindex()

Return the last occurrence of one text string within another

Format

`rindex(`*`text`*`,`*`string`*`)`

Parameters

Parameter	Definition
text	text to be examined for occurrences of *string* *The text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text.
string	one or more characters whose last occurrence is being identified within *text*

Description

The `rindex()` function returns the position of the last occurrence of *string* in *text* or 0 if *string* does not occur in *text*. Positions in *string* are numbered starting from one and are indicated as integers.

Example

The following is an example of the rindex() function:

```
sentence = "The quick brown fox jumped over the lazy dogs";
findme = "he";
print("The last occurrence is in position ",rindex(sentence,findme));
```

This example produces the following output:

```
The last occurrence is in position 34
```

BMC Software, Inc., Confidential and Proprietary Information

sec_store_get()

Return a value that resides in the secure area of a PATROL Agent

Format

`sec_store_get(`*`key`*`)`

Parameter

Parameter	Definition
`key`	text that identifies which value is returned by the secure area of the PATROL Agent

Description

The sec_store_get() function returns a value that has been stored in the Secure Key Store (SKS) archive. For more information about the SKS archive, see the *PATROL Security User Guide*.

You can use the *key* parameter to specify which value you want to retrieve from the agent. The *key* that you use must be the same one that was used to store the value. When sec_store_get() completes successfully, it returns the value in plain text. If it is not successful, sec_store_get() returns the NULL string, "". See the following table for a complete list of error messages.

As a security feature, the sec_store_get() function cannot retrieve a value unless the application that calls sec_store_get() has the proper permissions. When a value is stored by the sec_store_set() function, the *application_class* parameter(s) determine which applications can retrieve the value. An application must belong to a class that is specified by the *application_class* parameter to retrieve the value. For more information, see sec_store_set() on page 1-407.

BMC Software, Inc., Confidential and Proprietary Information

errno Value	Description of Failure
139	E_PSL_SKS_BAD_ACCOUNT The user account that executed this function does not have permission to access the SKS archive.
140	E_PSL_SKS_INSTANCE_ERR PATROL could not find the application instance that executed this function.
141	E_PSL_SKS_SET_ERR PATROL could not set this value in the SKS archive.
142	E_PSL_SKS_GET_ERR PATROL could not get this value from the SKS archive.

Example

This example demonstrates the sec_store_get() function.

```
sec_store_get("ORACLE/INSTANCE1/JSMITH");
```

sec_store_set()

Store and encrypt a text string in the PATROL Agent

Format

```
sec_store_set(key,value,[application_class, . . .])
```

Parameter

Parameter	Definition
key	text used by sec_store_get() to identify *value*
value	text that you want to put in the secure area of the PATROL Agent Often this function stores a password. In that case, the *key* is a user ID and *value* is a password.
application class	application class or classes that can retrieve *value* **Default** Application class that called the sec_store_set() function.

Description

The sec_store_set() function enters *value* into the Secure Key Store (SKS) archive. For more information about the SKS archive, see the *PATROL Security User Guide*.

Use this function to store and encrypt any text string. This function works with sec_store_get(), which you can use to retrieve a value stored by sec_store_set(). The sec_store_set() function returns 1 when successful and 0 when unsuccessful. See the following table for a complete list of error messages.

BMC Software, Inc., Confidential and Proprietary Information

errno Value	Description of Failure
139	E_PSL_SKS_BAD_ACCOUNT The user account that executed this function does not have permission to access the SKS archive.
140	E_PSL_SKS_INSTANCE_ERR PATROL could not find the application instance that executed this function.
141	E_PSL_SKS_SET_ERR PATROL could not set this value in the SKS archive.
142	E_PSL_SKS_GET_ERR PATROL could not get this value from the SKS archive.

The *application_class* parameter is a list of one or more application classes. To retrieve *value*, an application class must be on this list. If you do not specify an *application_class*, sec_store_set() uses the default for this parameter, which is the application class that called sec_store_set().

You can use this function to write a PSL script that performs an action that requires a password. For example, the script can use a password to open a database. In this scenario, *key* is a user ID and *value* is a password. The sec_store_set() function stores the password in the secure area of the PATROL Agent, and your script uses sec_store_get() to retrieve the password.

Giving a script password access to a protected resource presents a trade-off between security and usability. Requiring a person to enter a password is generally more secure than delegating this task to a script. The higher level of security, however, comes at a price to the person who has to remember the password or keep it in a safe place. While it is not the safest option, allowing a script to enter a password provides a much higher level of security than not using a password at all. Use this method of password management when you need to provide mid-level security.

BMC Software, Inc., Confidential and Proprietary Information

Example

This script demonstrates the sec_store_set() function.

```
sec_store_set("ORACLE/INSTANCE1/JSMITH", "paddy3",
   "PATROL_NT", "NT_SYSTEMS");
```

set()

Assign a value to a variable

Format

`set(variable,value)`

Parameters

Parameter	Definition
variable	the name of a variable in the PATROL Agent object hierarchy to which *value* is assigned The *variable* parameter is validated using the following rules: • *variable* is composed of name segments separated by a slash (/) • *variable* may start with a slash (/), but not end with a slash (/) • *variable* may not contain empty segments • *variable* may not contain unprintable characters • *variable* may not contain white space other than the space character • *variable* may contain spaces within a segment but not at the beginning or end of a segment
value	the numeric or string value that is assigned to *variable*

Description

The `set()` function sets the value of *variable* to be *value*. If *variable* is a relative name and does not exist in the context of the PSL script, the `set()` function successively searches each ancestor's context until *variable* is found or until the search fails in the context of the computer.

BMC Software, Inc., Confidential and Proprietary Information

The `set()` function always returns the NULL string. However, if the *variable* parameter is not valid, the PSL run-time error `532, “Badly formed object name.”` is generated.

Warning

The *value* parameter cannot exceed a numeric value of 2^{31-1}.

Example

The following PSL statement sets the value of RDB database Dev parameter MyParam to 10.

```
set("/RDB/Dev/MyParam/value",10);
```

set_alarm_ranges()

Set the range for the alarm conditions for a particular application or parameter

Format

```
set_alarm_ranges(new_ranges,param,appl,
      [param_oid],[path])
```

Parameters

Parameter	Definition
new_ranges	a newline-separated list of alarm range settings The first element of the list is the setting for the border alarm, the second element is for the first alarm zone (Alarm1), and the third element is for the second alarm zone (Alarm2). Each element of the list is in the following the format: *active min max trigger max_occ state* *active* = 1 or 0 indicating whether the alarm is active *min* = integer indicating the minimum of the zone *max* = integer indicating the maximum of the zone *trigger* = integer indicating how many times the parameter value must be in the range before triggering 0 immediately on alarm 1 after alarm has occurred *n* times 2 after all recovery actions fail *max_occ* = maximum number of times the alarm occurs *state* = 0(OK) or 1(WARN) or 2(ALARM) indicating what action occurs
param	the name of the parameter that is monitored
appl	either the name of the application or the application instance objectid
param_oid	optional parameter objectid
path	optional path to the parameter

Description

The `set_alarm_ranges()` function sets the range for the alarm conditions for a particular application or parameter.

Although *param* and *appl* are required, they can be "" if *param_oid* or *path* or both are supplied.

Calls to the `set_alarm_ranges()` function that specify the *param* and *appl* parameters without the *param_oid* and *path* parameters set the alarm ranges for the parameter class and affect all parameter instances of that class.

However, calls to the `set_alarm_ranges()` function that specify the *param_oid* and *path* parameters set the alarm ranges for that particular parameter instance and do not affect the alarm ranges for other instances of that class.

Return Value

The function returns the string "not found" if the parameter class cannot be found. Otherwise, it returns a number indicating which ranges were correctly set. The binary representation of the number should be considered. If the first bit is set, then the setting of the first range (border alarm range) has failed. If the second bit is set, the setting of the Alarm1 range has failed, and so on.

Note

The console's knowledge may not be a true reflection of the knowledge in the agent if this function is used. The consoles do gather alarm range information from the agent, but the agent sends it only when a parameter is drilled down on. Therefore, if the console has an open graph and the alarm ranges are subsequently changed by the `set_alarm_ranges()` function, then the range information displayed in the graph will not be updated until the graph is popped down and drilled into again.

Example

The following example changes the alarm ranges for the /NT_EVLOGFILES/NT_EVAPP/ELMEvFileFreeSpacePercent parameter, but does *not* change /NT_EVLOGFILES/NT_EVSEC/ELMEvFileFreeSpacePercent or /NT_EVLOGFILES/NT_EVSYS/ELMEvFileFreeSpacePercent.

```
new_ranges= "1 0 100 0 0 2\n1 5 10 0 0 1\n1 10 100 0 0 2";
result=set_alarm_ranges(new_ranges,"","","",
   "/NT_EVLOGFILES/NT_EVAPP/ELMEvFileFreeSpacePercent");
```

BMC Software, Inc., Confidential and Proprietary Information

The following example changes the alarm ranges for all three parameters: /NT_EVLOGFILES/NT_EVAPP/ELMEvFileFreeSpacePercent, /NT_EVLOGFILES/NT_EVSEC/ELMEvFileFreeSpacePercent, and /NT_EVLOGFILES/NT_EVSYS/ELMEvFileFreeSpacePercent.

```
new_ranges= "1 0 100 0 0 2\n1 5 10 0 0 1\n1 10 20 0 0 2";
result=set_alarm_ranges(new_ranges,"ELMEvFileFreeSpacePercent",
   "NT_EVLOGFILES");
```

The following example shows how to use the object ID to set the alarm ranges of a parameter:

```
new_ranges = "0 1 100 0 0 0\n0 80 90 0 0 0\n0 91 100 1 0 2";
oid = get("/NT_PHYSICAL_DISKS/DISK_0/PDpdDiskTimePercent/objectId");
printf("oid = %d\n", oid);
result = set_alarm_ranges(new_ranges, "", "", oid,
        "/NT_PHYSICAL_DISKS/DISK_0/PDpdDiskTimePercent");
```

set_locale()

Set or get the value of a locale category

Format

`set_locale(`*`category, locale`*`)`

Parameters

Parameter	Definition
category	locale categories Use ALL if you want to assign the default value to all locale categories. To use ALL, the *locale* parameter must be "". For an explanation of locale categories, see *PATROL Script Language Reference Manual Volume 1—PSL Essentials*. **Valid Values** ALL, CTYPE, MESSAGES (or MESSAGE), TIME, CODECVT (or IO, OS)
locale	locale name Use "" to assign the default value to the locale category. Use 0 to return the current value of the locale category. Use a specific locale name to assign that name to the locale category. For an explanation of locale names, see *PATROL Script Language Reference Manual Volume 1—PSL Essentials*. **Valid Values** "", 0, supported locale names: C, ja_JP.CP932, ja_JP.eucJP, ja_JP.SJIS, ko_KR.eucKR, zh_CN.gb, zh_TW.big5, zh_TW.eucTW **Default Value** same as CTYPE locale category

BMC Software, Inc., Confidential and Proprietary Information

Description

The set_locale() function sets or gets the value of a locale category.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

The value of the *locale* parameter determines whether set_locale() sets or gets a value. If you enter a specific locale name for *locale*, set_locale() tries to assign this locale name to the locale category specified by the *category* parameter. The set_locale() function returns 1 if it sets the category and 0 if it fails to set the value. If *locale* is 0, set_locale() returns the current value of *category*. If *locale* is "", set_locale() assigns the default value to *category*.

If you use ALL as *category*, you are specifying all locale categories. You can use ALL with *locale* set only to "", which sets all categories to their default values.

If *category* or *locale* is invalid, set_locale() sets the *errno* variable to 134 (E_PSL_SET_LOCALE_FAILED) and returns NULL.

Note

You cannot change the value of the CTYPE locale category.

Example

If a PATROL Agent is running under the ja_JP.eucJP locale, the following script shows how to read an SJIS-coded file and a eucJP-coded file.

```
set_locale("CODECVT", "ja_JP.SJIS");
data = cat("/home/users/text.sjis");
print(data);
set_locale("CODECVT", "");
data = cat("/home/users/text.euc");
print(data);
```

BMC Software, Inc., Confidential and Proprietary Information

share()

Convert a local channel into a shared global channel

Format

`share(`*`channel`*`,`*`name`*`)`

Parameters

Parameter	Definition
channel	process I/O channel number that was returned when the channel was opened using the fopen() or popen() function
`name`	character string used to identify the shared channel in the table of global channels BMC Software recommends that you specify a nonnumeric *name* to avoid conflicts with numbers used internally for local channels. Using a number for *`name`* does not actually cause the share() function to fail but will raise a PSL run-time warning. The share() function will dutifully place the specified numeric *`name`* in the global table, leading to potential conflicts with local channels in close(), read(), write(), and readln() functions.

Description

The share() function is the main function for using shared channels. The share() function propagates an existing local *channel* into the table of global channels as *name*. Channels opened by either the popen() or fopen() functions can be shared.

If the share() function is successful, it returns 0. The local channel is no longer available in the process that opened it and does not require a close() function. In fact, the close() function will return an error since it will not find the local channel.

BMC Software, Inc., Confidential and Proprietary Information

The share() function will fail, returning –1 and setting the PSL errno variable under the following conditions:

- the local channel does not exist
- the global channel *name* already exists in the global channel table

Upon failure, the local channel is unchanged and still available. No global channel is added.

A global channel is referred to by `name` when passed to the read(), readln(), write(), and close() functions. These functions will first search the local channel table containing only channel numbers and then the global channel table.

The share() function and the Knowledge Module Developer

The PATROL Agent performs no explicit cleanup of the shared channels. It is up to the Knowledge Module developer to clean up as well as possible. Unfortunately, there is no easy way to intercept a console request to deactivate an application or parameter and hence doing so for an application using shared channels will most likely *not* terminate the external process.

For a brute-force cleanup, it is possible to use close(*name*,3) to remove a global channel and terminate its external process. It is also possible to write a “cleanup all shared channels” menu command using the get_chan_info() function to return a list of shared channels and close() to remove them.

Example

The following PSL script uses the share() function to convert a local channel into a shared global channel:

```
chan = fopen("/etc/passwd"."r");
share(chan,"passwd_handle");
# read from the global channel
while ((user = readln("passwd_handle")) != EOF) {
   printf("%s\n",user);
}
```

BMC Software, Inc., Confidential and Proprietary Information

sin()

Return the sine of the argument

Format

`sin(`*`radians`*`)`

Parameter

Parameter	Definition
`radians`	arc length in radians whose sine is to be determined **Valid Values** $-\infty \leq radians \leq \infty$

Description

The sin() function returns the sine of *radians*. The output range for the sin() function is $-1 \leq \sin() \leq 1$.

Example

The following example prints a table of the sin() function in the range 0 to 2π in increments of 30 degrees (instead of radians):

```
function main() {
   PI = 3.141593;
   print("SINE FUNCTION TEST\n\n");
   print("   degrees  calculated  expected  exact value\n");
   print("   -------  ----------  --------  -----------\n");

   # use a PSL list to hold the expected values (The printf
   # statement will evaluate the elements containing sqrt
   # expressions)

   expected = [
      0,0.5,0.5 * sqrt(3),
      1,0.5 * sqrt(3),0.5,
      0,-0.5,-0.5 * sqrt(3),
      -1,-0.5 * sqrt(3),0.5,0
   ];

   # Use a PSL List to hold the exact value strings

   exact = [
      " 0"," 0.5"," 0.5 * sqrt(3)",
      " 1"," 0.5 * sqrt(3)"," 0.5",
      " 0","-0.5","-0.5 * sqrt(3)",
      "-1","-0.5 * sqrt(3)","-0.5"," 0"
   ];

   increment = 0;
   while (increment <= 360) {
      printf(
         "     %3d     %8.5f   %8.5f  %s\n",
         increment,
         sin(2 * PI * increment / 360),
         trim(nthline(expected,int(1 + increment / 30)),"\n"),
         trim(nthline(exact,int(1 + increment / 30)),"\n")
      );
      increment += 30;
   }
   return;
}
```

BMC Software, Inc., Confidential and Proprietary Information

This example produces the following output:

```
SINE FUNCTION TEST

   degrees  calculated  expected  exact value
   -------  ----------  --------  -----------
       0      0.00000    0.00000   0
      30      0.50000    0.50000   0.5
      60      0.86603    0.86603   0.5 * sqrt(3)
      90      1.00000    1.00000   1
     120      0.86603    0.86603   0.5 * sqrt(3)
     150      0.50000    0.50000   0.5
     180      0.00000    0.00000   0
     210     -0.50000   -0.50000  -0.5
     240     -0.86603   -0.86603  -0.5 * sqrt(3)
     270     -1.00000   -1.00000  -1
     300     -0.86603   -0.86603  -0.5 * sqrt(3)
     330     -0.50000    0.50000  -0.5
     360      0.00000    0.00000   0
```

BMC Software, Inc., Confidential and Proprietary Information

sinh()

Return the hyperbolic sine of the argument

Format

`sinh(`*`argument`*`)`

Parameter

Parameter	Definition
`argument`	numeric value whose hyperbolic sine is to be determined **Valid Values** $-\infty \leq argument \leq \infty$

Description

The sinh() function returns the hyperbolic sine of *argument*. The hyperbolic sine is defined by the expression:

$$\sinh(x) = (e^{x} - e^{-x})/2$$

where e is the base for the natural logarithms (e =2.71828 . . .). The output range for the sinh() function is $-\infty \leq \sinh() \leq \infty$.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL script uses the sinh() function to verify the hyperbolic function power identity $\sinh^2 x = 0.5(\cosh 2x - 1)$:

```
function main() {
   print("HYPERBOLIC SINE TEST\n\n");
   print("   value   sinh(x)   pow(sinh(x),2)  0.5 * (cosh(2x) -1)\n");
   print("   -----  ---------  --------------  -------------------\n");
   count = -2;
   while (count <= 2) {
      printf(
         "    %+3d  %+9.5f     %+9.5f           %+9.5f\n",
         count, result = sinh(count),
         pow(result,2),
         0.5 * (cosh(2 * count) - 1)
      );
      count += 0.1;
   }
   return;
}
```

This example produces the following output:

```
HYPERBOLIC SINE TEST

   value   sinh(x)   pow(sinh(x),2)  0.5 * (cosh(2x) -1)
   -----  ---------  --------------  -------------------
     -2   -3.62686     +13.15411           +13.15412
     -1   -3.26816     +10.68089           +10.68089
     -1   -2.94217      +8.65639            +8.65639
     -1   -2.64563      +6.99937            +6.99937
     -1   -2.37557      +5.64332            +5.64332
     -1   -2.12928      +4.53383            +4.53383
     -1   -1.90430      +3.62637            +3.62636
     -1   -1.69838      +2.88450            +2.88450
     -1   -1.50946      +2.27847            +2.27847
     -1   -1.33565      +1.78395            +1.78395
     -1   -1.17520      +1.38110            +1.38110
     +0   -1.02652      +1.05374            +1.05374
     +0   -0.88811      +0.78873            +0.78873
     +0   -0.75858      +0.57545            +0.57545
     +0   -0.63665      +0.40533            +0.40533
     +0   -0.52109      +0.27154            +0.27154
     +0   -0.41075      +0.16872            +0.16872
     +0   -0.30452      +0.09273            +0.09273
     +0   -0.20134      +0.04054            +0.04054
     +0   -0.10017      +0.01003            +0.01003
     +0   +0.00000      +0.00000            +0.00000
     +0   +0.10017      +0.01003            +0.01003
     +0   +0.20134      +0.04054            +0.04054
     +0   +0.30452      +0.09273            +0.09273
     +0   +0.41075      +0.16872            +0.16872
     +0   +0.52109      +0.27154            +0.27154
     +0   +0.63665      +0.40533            +0.40533
     +0   +0.75858      +0.57545            +0.57545
     +0   +0.88811      +0.78873            +0.78873
     +0   +1.02652      +1.05374            +1.05374
     +1   +1.17520      +1.38110            +1.38110
     +1   +1.33565      +1.78395            +1.78395
     +1   +1.50946      +2.27847            +2.27847
     +1   +1.69838      +2.88450            +2.88450
     +1   +1.90430      +3.62637            +3.62636
     +1   +2.12928      +4.53383            +4.53383
     +1   +2.37557      +5.64332            +5.64332
     +1   +2.64563      +6.99937            +6.99937
     +1   +2.94217      +8.65639            +8.65639
     +1   +3.26816     +10.68089           +10.68089
     +2   +3.62686     +13.15411           +13.15412
```

BMC Software, Inc., Confidential and Proprietary Information

sleep()

Suspend process execution for a number of seconds

Format

`sleep(`*`seconds`*`)`

Parameter

Parameter	Definition
`seconds`	integer specifying the number of seconds the process should be suspended **Valid Values** • *seconds* > 0 is the number of seconds the process will sleep • *seconds* ≤ 0 the timer expires immediately

Description

The sleep() function suspends a PSL process for the specified number of seconds. While suspended, the PSL process consumes no CPU resources and is not interpreted until awakened by the expiration of the *seconds* timer.

Note

The sleep() function only suspends the process that calls it. All other PSL processes continue normal execution.

The sleep() function returns a run-time warning if *seconds* is nonnumeric, in which case the timer defaults to zero.

Example

The following is an example of the sleep() function:

```
print("Hello world!\n");
sleep(10);
print("Good-bye world!\n");
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_agent_config()

Indicate if the PATROL SNMP Sub-Agent is active or not active

Format

```
snmp_agent_config()
```

Parameters

This function does not accept any parameters.

Description

The snmp_agent_config() function returns one of the following messages indicating if the PATROL SNMP Sub-Agent is active or inactive.

```
SNMP support is not active.
```

or

```
SNMP support is active.
```

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example uses the snmp_agent_config() function to create a user-defined function, verify_snmp_agent(), to check for SNMP support.

The main() section of the example starts SNMP, checks for SNMP support, stops SNMP, and then checks again for SNMP support using the user-defined verify_snmp_agent() function:

```
function verify_snmp_agent() {
   print("Checking for SNMP support...  ");
#Check the result of snmp_agent_config()function for the text "not" starting
#in the 17th character of the output string using the substr() fubction.
   if (substr(snmp_agent_config(),17,3)=="not") {
      print("SNMP NOT ACTIVE\n");
   } else {
      print("SNMP ACTIVE\n");
      }
   return;
}
function main() {
   print("Starting SNMP\n");
   snmp_agent_start();        #Starting SNMP
   verify_snmp_agent();       #Checking for SNMP support
   print("Stopping SNMP\n");
   snmp_agent_stop();         #Stopping SNMP
   verify_snmp_agent();       #Checking for SNMP support
}
```

This example produces the following output:

```
Starting SNMP
Checking for SNMP support...  SNMP ACTIVE
Stopping SNMP
Checking for SNMP support...  SNMP NOT ACTIVE
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_agent_start()

Start the PATROL SNMP Sub-Agent

Format

```
snmp_agent_start()
```

Parameters

This function does not accept any parameters.

Description

The snmp_agent_start() function starts the PATROL SNMP Sub-Agent listening for SNMP requests. If the snmp_agent_start() function is issued and the PATROL SNMP Sub-Agent is already running, the snmp_agent_start() function has no effect.

The snmp_agent_start() function returns the string `OK` if successful, or the string `ERR` if an SNMP master agent is not running.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example checks the output from snmp_agent_config() function for the text "SNMP support is not active." If the text is found, SNMP support is not active, and the snmp_agent_start() function is used to start the PATROL SNMP Sub-Agent. The return value of the snmp_agent_start() function is checked for the string `OK` to verify that the PATROL SNMP Sub-Agent started:

```
if (grep("SNMP support is not active.",snmp_agent_config())) {
        if (snmp_agent_start() == "OK")
        {
            print("Patrol SNMP sub-agent started.\n");
        }
        else
        {
            print("Patrol SNMP sub-agent failed to start.\n");
            print("Make sure the SNMP Master Agent is running.\n");
        }
    }
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_agent_stop()

Stop the PATROL SNMP Sub-Agent

Format

```
snmp_agent_stop()
```

Parameters

This function does not accept any parameters.

Description

The snmp_agent_stop() function stops the PATROL SNMP Sub-Agent if it was running or performs no action if it was not. The snmp_agent_stop() function returns the string `OK` if it stops the PATROL SNMP Sub-Agent or finds it not running.

Example

The following example checks the output from snmp_agent_config() function for the text "not" starting in the 17th position. If the text "not" is not found, SNMP support is active, and the snmp_agent_stop() function is used to stop the PATROL SNMP Sub-Agent:

```
if (substr(snmp_agent_config(),17,3)!="not") {
snmp_agent_stop();
print("Patrol SNMP sub-agent stopped\n");
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_close()

Close a PATROL Agent SNMP session

Format

`snmp_close(`*`session`*`)`

Parameter

Parameter	Definition
`session`	the ID of the SNMP session that is to be closed The *session* is the value returned when an snmp_open() function is executed.

Description

The snmp_close() function closes *session* with the PATROL SNMP Sub-Agent that was opened with the snmp_open() function and removes internal structures belonging to *session*. The snmp_close() function sets the PSL variable errno = 93 (E_PSL_NO_SUCH_ID) if *session* is not a valid SNMP session ID. The snmp_close() function returns the string “OK” if successful, or the NULL string otherwise.

Example

The following example opens an SNMP session makes a query to determine what SNMP agent is listening on port 8161 and closes the SNMP session:

```
session=snmp_open(“runner”,””,””,””,””,”8161”);
print(snmp_get(session,”.1.3.6.1.2.1.1.1.0”));
snmp_close(session);
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_config()

Return the default parameters for one or more open PATROL SNMP sessions that were opened with the snmp_open() function

Format

```
snmp_config()
snmp_config(session)
snmp_config(session,community,timeout,retries)
```

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
`session`	the ID of the SNMP session whose default parameters are to be returned or set The *session* is the ID returned when an snmp_open() function is executed.
`community`	character string used as a community string for SNMP operations **Default** no change to existing *community*
`timeout`	integer value of the session timeout parameter in milliseconds (ms) **Default** no change to exiting *timeout*
`retries`	integer value indicating the number of retries to attempt when a timeout occurs **Default** no change to exiting *retries*

Description

The snmp_config() function has three different formats that perform three distinct actions:

- The format snmp_config() returns a list of all open SNMP session separated by new-line characters. If no sessions are open the NULL string is returned. The following is an example of the snmp_config() function output for three active sessions:

  ```
  sess1

  sess2

  sess3
  ```

- The format snmp_config(*session*) returns a string of the default parameters for *session* separated by spaces.

  The output string for each session is as follows:

  host community timeout retries

Value	Definition
host	the TCP/IP address of the computer system to which the session is opened
community	name of community string used for SNMP operations
timeout	session timeout value in milliseconds (ms)
retries	number of retries to attempt when a timeout occurs

- The format snmp_config(*session,community,timeout,retries*) sets the default parameters of *session* to the indicated values. If any parameter is specified in the snmp_config() function as the NULL string(""), the existing default value for the parameter is unchanged. Returns the string `OK` if successful or set the errno variable to 93 indicating a bad session id.

Example

The following example includes each of the three uses of the snmp_config() function (listing existing sessions, listing session settings, and setting session settings):

```
# Open three snmp sessions

    snmp_sessiona = snmp_open("runner","","","","","");
    snmp_sessionb = snmp_open("gsmithisdn","","","1000","6","161");
    snmp_sessionc = snmp_open("gsmith","","","","","8161");

    # Print session id for each of the sessions

    print("snmp_sessiona: ".snmp_sessiona."\n");
    print("snmp_sessionb: ".snmp_sessionb."\n");
    print("snmp_sessionc: ".snmp_sessionc."\n");
    print("\n");

    # Capture and print list of open snmp sessions ignoring blank lines

    sessions = grep("^$",snmp_config(),"v");
    print("Output from snmp_config():\n".sessions."\n");
```

BMC Software, Inc., Confidential and Proprietary Information

```
print("Number of open sessions: ".lines(sessions)."\n");
print("\n");

# Loop through list of open snmp sessions and print snmp_config for each

foreach session (sessions)
{
    print("Output from snmp_config for ".session.": ".snmp_config(session)."\n");
}
print("\n");

# Change default snmp session parameters for sessiona.  Change timeout to 700
# and retries to 5.

print("Use snmp_config() to change default parameters for ".snmp_sessiona.": ");
print(snmp_config(snmp_sessiona,"",800,5)."\n");

# Change default snmp session parameters for sessionb.  Change community to
# "internal", timeout to 600, and retries to 7.

print("Use snmp_config() to change default parameters for ".snmp_sessionb.": ");
print(snmp_config(snmp_sessionb,"internal",600,7)."\n");
print("\n");

# Loop through list of open snmp sessions and print snmp_config for each
# then close the snmp session.

foreach session (sessions)
{
    print("Output from snmp_config for ".session.": ".snmp_config(session)."\n");
    snmp_close(session);
}
```

This example produces the following output:

```
snmp_sessiona: sess21
snmp_sessionb: sess22
snmp_sessionc: sess23

Output from snmp_config():
sess21
sess22
sess23

Number of open sessions: 3

Output from snmp_config for sess21: 172.19.201.36 public 500 3
Output from snmp_config for sess22: 172.21.1.223 public 1000 6
Output from snmp_config for sess23: 172.19.1.13 public 500 3

Use snmp_config() to change default parameters for sess21: OK
Use snmp_config() to change default parameters for sess22: OK

Output from snmp_config for sess21: 172.19.201.36 public 800 5
Output from snmp_config for sess22: 172.21.1.223 internal 600 7
Output from snmp_config for sess23: 172.19.1.13 public 500 3
```

BMC Software, Inc., Confidential and Proprietary Information

_snmp_debug()

Activate SNMP debugging features

Format

`_snmp_debug(`*`flags`*`)`

Parameter

Parameter	Definition
`flags`	binary flags that activate PSL SNMP debugging features **Valid Values** 0 deactivate debug reporting 1 dump all input/output packets to the PATROL computer console display window 2 provide additional information about timeout errors 3 apply both ranges 1 and 2

Description

The _snmp_debug() function enables the basic SNMP debugging function within PSL. The _snmp_debug() function returns the previous *flags* value or NULL indicating that the new *flags* resulted in an error.

Example

The following is an example of the _snmp_debug() function:

```
switch (debug_level) {
   case 1: {new_level = _snmp_debug(1);}   # dump I/O packets to the console
   case 2: {new_level = _snmp_debug(2);}   # provide more timeout error info
   case 3: {new_ level = _snmp_debug(3);}  # both levels 1 and 2
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_get()

Request SNMP variables from the PATROL SNMP Management Information Base (MIB) on the agent that the SNMP session is opened with

Format

`snmp_get(`*`session`*`,[`*`variable1`*`,...`*`variablen`*`])`

Parameters

Parameter	Definition
`session`	the ID of the SNMP session to which the SNMP get-request command is submitted The *session* is the ID returned when an snmp_open() function is executed.
`variablen`	name of an SNMP MIB variable The *variablen* can be specified in numeric form or in symbolic form if the MIB is known to the PATROL Agent. One variable name is required; additional variable names are optional.

Description

The snmp_get() function sends an SNMP get-request command to *session* to retrieve the values of *variable1*, . . . *variablen* from the SNMP Management Information Base (MIB). The snmp_get() function returns a space-separated list of the variables or the NULL string if an error occurs.

The snmp_get() function can set the PSL errno value to one of the following values:

errno Value	Description
95	E_PSL_TIMEOUT the request timed out
93	E_PSL_NO_SUCH_ID a bad session id or no session ID
100	E_PSL_SNMP_ERROR an error is returned in the response packet from the SNMP session

Output Format

The output of the snmp_get() function depends on the data type of the parameters that are used to call this function. Using string parameters returns information in the following format:

name length type value

Field	Definition
name	name of an SNMP MIB variable
length	length of *value*
type	character string indicating the *name* type
value	value of *name*

Calling the snmp_get() function with parameters of any other data type returns information in the following format:

name type value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the *name* type
value	value of *name*

Example

The following PSL script uses the snmp_get() function to return a value from the PATROL SNMP MIB:

```
session = snmp_open("runner","","","","","");
    value = snmp_get(session,".1.3.6.1.4.1.1031.1.1.1.6.1.1.0");
    if (errno == 0)
    {
        print("snmp_get() for PatrolAgent <platform>.km:\n");
        print(value);
    }
    else
    {
        print("snmp_get() failed...  errno = ".errno.".\n");
    }
    snmp_close(session);
```

The previous example produces the following output:

```
snmp_get() for PatrolAgent <platform>.km:
1.3.6.1.4.1.1031.1.1.1.6.1.1.0        2  string       HP
```

snmp_get_next()

Return variables from the SNMP Management Information Base (MIB) that are lexicographically next to the specified variable names on the agent that the SNMP session is opened with

Format

`snmp_get_next(`*`session`*`,`*`variable1`*`,[...`*`variablen`*`])`

Parameters

Parameter	Definition
`session`	the ID of the SNMP session to which the SNMP get-next-request is submitted The *session* is the ID returned when an snmp_open() function is executed.
`variablen`	the variable name that is to receive a value from the SNMP MIB for session The string *variablen* is used to find the variable in the MIB whose name is the nearest alphabetically greater name than *variablen*. The value of that variable is returned. One variable name is required; additional variable names are optional.

Description

The snmp_get_next() sends an SNMP get-next-request to *session* to retrieve the value of the Management Information Base (MIB) variable whose name is lexicographically next to the name *variablen*. The snmp_get_next() function returns a space-separated list of the variables and their MIB names or the NULL string if an error occurs.

The snmp_get_next() function can set the PSL errno value to one of the following values:

errno Value	Description
95	E_PSL_TIMEOUT the request timed out
93	E_PSL_NO_SUCH_ID a bad session id or no session ID
100	E_PSL_SNMP_ERROR an error is returned in the response packet from the SNMP session

Output Format

The output of the snmp_get_next() function is a s list of SNMP variables in the following format:

name type length value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the `name` type
length	length of *value*
value	value of *name*

Example

The following example uses the snmp_get_next() function to retrieve and print all the application instances in the PATROL MIB. The application instance names can also be retrieved (much more easily) using the snmp_walk() function (see “snmp_walk()” on page 1-479):

```
function main() {
   local myhost,session,app_start,MIB_entry,address;
myhost = get("/hostname");
   session = snmp_open(myhost,"1","","","",8161);
   app_start = ".1.3.6.1.4.1.1031.1.1.1.7.1.1";
   MIB_entry = trim(snmp_get_next(session,app_start),"\n");
   address = nthargf(MIB_entry,1," ");
   while (nthargf(MIB_entry,14,".") == "applInstName") {
      printf("%s\n",MIB_entry);
      MIB_entry = trim(snmp_get_next(session,address),"\n");
      address = nthargf(MIB_entry,1," ");
   }
   printf("\n");
   snmp_close(session);
   return;
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_h_get()

Return variables from the PATROL SNMP Management Information Base (MIB) using a brief (local) session and default parameters from a specified agent

Format

`snmp_h_get(`*`host`*`,`*`variable1`*`,[...`*`variablen`*`])`

Parameters

Parameter	Definition
`host`	name of the computer to which the SNMP get-request is submitted The host must be a name known to the local computer in the **/etc/hosts** file, the Domain Name System (DNS), the Network Information Service (NIS), or an IP address.
`variablen`	name of an SNMP MIB variable The variablen can be specified in numeric form or in symbolic form if the MIB is known to the PATROL Agent. One variable name is required; additional variable names are optional.

Description

The snmp_h_get() function is the equivalent of the following series of PSL function calls:

```
session = snmp_open(host);
snmp_get(session,variable1,...,variablen);
snmp_close(session);
```

The snmp_h_get() function performs the following tasks:

- opens an SNMP session on host. (Refer to the snmp_open() function description on page 1-456 for the default parameters used to open the session.)

- sends an SNMP get-request command to the session to retrieve the values of *variable1, . . . variablen* from the SNMP Management Information Base (MIB)

- closes the SNMP session

The snmp_h_get() function returns a space-separated list of the variables or the NULL string if an error occurs.

The snmp_h_get() function can set the PSL errno value to one of the following values:

errno Value	Description
95	E_PSL_TIMEOUT the request timed out
100	E_PSL_SNMP_ERROR an error is returned in the response packet from the SNMP session

Output Format

The output of the snmp_h_get() function is a s list of SNMP variables in the following format:

name type length value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the `name` type
length	length of *value*
value	value of *name*

Example

The following example uses the snmp_h_get() function to retrieve the first application instance name from the PATROL MIB:

```
function main() {
   local myhost,param_start;
   myhost = get("/hostname");
   param_start = ".1.3.6.1.4.1.1031.1.1.1.6.1.1.0";
   printf("%s\n",trim(snmp_h_get(myhost,param_start),"\n"));
   return;
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_h_get_next()

Return variables from the PATROL SNMP Management Information Base (MIB) that are lexicographically next to the specified variable names using a brief (local) session and default parameters

Format

```
snmp_h_get_next(host,variable1,[...variablen])
```

Parameters

Parameter	Definition
`host`	name of the computer to which the SNMP get-next-request is submitted The *host* must be a name known to the local computer in the **/etc/hosts** file, the Domain Name System (DNS), the Network Information Service (NIS), or an IP address.
`variablen`	variable name which is to receive a value from the SNMP MIB for session The string *variablen* is used to find the variable in the MIB whose name is the nearest alphabetically greater name than *variablen*. The value of that variable is returned. One variable name is required; additional variable names are optional.

Description

The snmp_h_get_next() function is the equivalent of the following series of PSL function calls:

```
session = snmp_open(host);
snmp_get(session,variable1,...,variablen);
snmp_close(session);
```

The snmp_h_get_next() function performs the following tasks:

- opens an SNMP session on *host*. (Refer to the snmp_open() function description on page 1-456 for the default parameters used to open the session.)
- sends an SNMP get-next-request command to the session to retrieve the value of the Management Information Base (MIB) variable whose name is lexicographically next to the name *variablen*
- closes the SNMP session

The snmp_h_get_next() function returns a space-separated list of the variables and their MIB names or the NULL string if an error occurs.

The snmp_h_get_next() function can set the PSL errno value to one of the following values:

errno Value	Description
95	E_PSL_TIMEOUT the request timed out
100	E_PSL_SNMP_ERROR an error is returned in the response packet from the SNMP session

Output Format

The output of the snmp_h_get_next() function is a s list of SNMP variables in the following format:

name type length value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the `name` type
length	length of *value*
value	value of *name*

Example

The following example uses the snmp_h_get_next() function to retrieve the second application instance name from the PATROL MIB:

```
function main() {
   local myhost,param;
   myhost = get("/hostname");
   param = ".1.3.6.1.4.1.1031.1.1.1.6.1.1.0";
   printf("%s\n",trim(snmp_h_get_next(myhost,param),"\n"));
   return;
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_h_set()

Open a brief PATROL SNMP session with default parameters to set variables in the SNMP Management Information Base (MIB)

Format

```
snmp_h_set(host,objectid1 type1 value1,
[...objectidn typen valuen])
```

Parameters

Parameter	Definition
`host`	name of the computer to which the SNMP set-request is submitted The *host* must be a name known to the local computer in the **/etc/hosts** file, the Domain Name System (DNS), the Network Information Service (NIS), or an IP address.
`objectidn`	character string SNMP object identifier for the variable whose value is to be changed
`typen`	character string indicating the *objectidn* type **Valid Values** integer string
`valuen`	value assigned to *objectidn*

BMC Software, Inc., Confidential and Proprietary Information

Description

The snmp_h_set() function is the equivalent of the following series of PSL function calls:

```
session = snmp_open(host);
snmp_set(session,objectid1 type1 value1,...,
      objectidn typen valuen);
snmp_close(session);
```

The snmp_h_set() function performs the following tasks:

- opens an SNMP session on *host*

  (See the snmp_open() function on page 1-456 for the default parameters used to open the session.)

- sends an SNMP set-request to the session to set specified variables in the Management Information Base (MIB)
- closes the SNMP session

The three parts of the snmp_h_set() function variable replacement, *objectidn*, *typen*, and *valuen* must be separated by spaces.

The snmp_h_set() function returns the new value of the MIB variable, or when an error occurs, the NULL string or an error message.

MIB variable value

The snmp_h_set() function returns the new value of an SNMP MIB variable in the following format:

name type length value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the *name* type
length	length of the *value*
value	value of *name*

BMC Software, Inc., Confidential and Proprietary Information

When the snmp_h_set() function fails to set the PATROL SNMP MIB variable it can return the NULL and set the PSL errno to one of the following values:

errno Value	Description
0	E_PSL_NO_ERROR • invalid string length • no parameter • redundancy of snmp_set with the same parameter • multiple objects using the same oid • using set to register an snmp manager
95	E_PSL_TIMEOUT the request timed out
96	E_PSL_BAD_FUNCTION_PARAMETER • bad object id • invalid object type • no object type • no object value
93	E_PSL_NO_SUCH_ID • bad session id • no session ID
100	E_PSL_SNMP_ERROR multiple parameters with different oids

error message

When the snmp_h_set() function fails to set the PATROL SNMP MIB variable because of an invalid *objectid* the following message is returned:

```
There is no such variable name in this MIB.
```

Example

The following example uses the snmp_h_set() function to set a value in the PATROL MIB:

```
if (snmp_h_get(myhost,MIBaddr)) {
   snmp_h_set(myhost,MIBaddr . " " . newtype . " " . newvalue);
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_open()

Open a PATROL Agent SNMP session with a PATROL Agent

Format

`snmp_open(`*`host`*`,`*`version`*`,`*`community`*`,`*`timeout`*`,`*`retries`*`,`*`port`*`)`

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
`host`	name of the computer to open a session with the SNMP agent The *host* must be a name known to the local computer in the **/etc/hosts** file, the Domain Name System (DNS), the Network Information Service (NIS), or an IP address.
`version`	the SNMP version **Valid Values** 1 (this function does not currently support SNMP Version 2)
`community`	name of the character string used as a community string for SNMP operations **Default** the value of the **/snmp/default_r_community** agent configuration variable or public
`timeout`	the session timeout parameter in milliseconds (ms) **Default** the value of the **/snmp/default_timeout** agent configuration variable or 500 ms (0.5 second)
`retries`	the number of retries to attempt when a timeout occurs **Default** the value of the **/snmp/default_retries** agent configuration variable or 3
`port`	user Datagram Protocol (UDP) port number on which to open the SNMP session **Default** the value of the **/snmp/default_port** agent configuration variable or 8161

Description

The snmp_open() function opens a session to the SNMP agent on *host* using User Datagram Protocol *port*, specifying the indicated session parameters. The snmp_open() function returns the session id if successful and the NULL string if unsuccessful.

The session ID consists of the characters “sess” followed by a numeric value as follows:

```
sess1
```

The snmp_open() function does not verify that it can reach the host or that an SNMP agent exists on host—it just creates its internal data structure with the parameters to be used in SNMP transactions.

All sessions to an SNMP agent share a single socket opened by the first snmp_open() function call and closed by the last snmp_close() function call. The snmp_open function sets the PSL errno variable to E_PSL_SOCKET_BUSY if the call is the first call to open a session and the socket cannot be opened. The condition may indicate that the SNMP agent has no more available file descriptors.

Example

The following PSL script presents a common way to open a session with the PATROL SNMP Agent service on your local machine:

```
myhost = get("/hostname");
session = snmp_open(myhost,"","","","",8161);
```

snmp_set()

Set variables in the PATROL SNMP Management Information Base (MIB)

Format

```
snmp_set(session,
      objectid1 [lengthn] type1 value1,
      [...,objectidn lengthn typen valuen])
```

Parameters

Parameter	Definition
`session`	the ID of the SNMP session to which the SNMP set-request is submitted The *session* is the ID returned when an snmp_open() function is executed.
`objectidn`	character string SNMP object identifier for the variable whose value is to be changed
`lengthn`	optional integer used for *typen* = character to indicate the length of the character string This parameter is provided to support multiline strings. The snmp_set() function will return an error if the text string *valuen* is not greater than or equal to *lengthn* characters.
`typen`	character string indicating the objectidn type **Valid Values** integer string
`valuen`	value to be assigned to the MIB variable *objectidn*

BMC Software, Inc., Confidential and Proprietary Information

Description

The snmp_set() function sends an SNMP set-request to *session* to set specified variables in the Management Information Base (MIB).

The snmp_set() function requires that the SNMP session community property be "private." The community property is set when the SNMP session is opened with the snmp_open() function.

The three parts of the snmp_set() function variable replacement, *objectidn*, *typen*, and *valuen* must be separated by spaces.

Return Values

The snmp_set() function returns the new value of the MIB variable, or when an error occurs, the NULL string or an error message.

MIB variable value

The snmp_set() function returns the new value of an SNMP MIB variable in the following format:

name type length value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the *name* type
length	length of the *value*
value	value of *name*

errno variable

When the snmp_set() function fails to set the PATROL SNMP MIB variable, it can return the NULL and set the PSL errno to one of the following values:

errno Value	Description
0	E_PSL_NO_ERROR • invalid string length • no parameter • redundancy of snmp_set with the same parameter • multiple objects using the same oid • using set to register an snmp manager
95	E_PSL_TIMEOUT the request timed out
96	E_PSL_BAD_FUNCTION_PARAMETER • bad object id • invalid object type • no object type • no object value
93	E_PSL_NO_SUCH_ID • bad session id • no session ID
100	E_PSL_SNMP_ERROR multiple parameters with different oids

When the snmp_set() function fails to set the PATROL SNMP MIB variable because of an invalid objectid, length, or type, one of the following messages is returned:

```
There is no such variable name in this MIB.
The value given has the wrong type or length.
```

Example

The following snmp_set() function will set the SNMP variable .1.3.4.6.4.1.0 to the string “some_text_is_25\nchar_long”:

```
snmp_set(sess,".1.3.4.6.4.1.0 25 string some_text_is_25\nchar_long");
```

snmp_trap_ignore()

Stop accumulating incoming PATROL SNMP event traps

Format

```
snmp_trap_ignore()
```

Parameters

This function does not accept any parameters.

Description

The snmp_trap_ignore() function ends the accumulation of incoming SNMP event traps by interrupting any active snmp_trap_receive() function and closing the socket for event-trapping that was opened with the snmp_trap_listen() function.

The snmp_trap_ignore() function returns the character string OK if successful or the NULL string otherwise.

Example

The following PSL statements cause the PATROL SNMP Agent service to ignore incoming traps:

```
opstatus = snmp_trap_ignore();
if (opstatus == "") {
   # snmp_trap_ignore() was bad!
   # diagnosis and recovery statements here
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_trap_listen()

Start accumulating incoming PATROL SNMP event traps

Format

```
snmp_trap_listen()
```

Parameters

This function does not accept any parameters.

Description

The snmp_trap_listen() opens a socket to accumulate incoming event traps to the SNMP agent. The socket number is chosen in the following order:

1. value of the **/snmp/trap_port** variable
2. the **snmp-trap/udp** service
3. 162

The snmp_trap_listen() function does not block the process from which it is called. The snmp_trap_listen() function returns the characters `OK` if it is successful and the NULL string if it is not. The snmp_trap_listen() function can also set the PSL errno variable as follows:

errno Value	Description
98	E_PSL_SNMP_ALREADY_LISTENING a socket is already active to accumulate SNMP traps
94	E_PSL_SOCKET_BUSY the socket that the snmp_trap_listen() function is attempting to use a socket that is already in use

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL script is uses the snmp_trap_ignore(), snmp_trap_listen(), and snmp_trap_receive() functions:

```
# Stop accumulating traps.  Clears any snmp_trap_listen().

    snmp_trap_ignore();

    # Start listening for (accumulating) traps

    ret = snmp_trap_listen();
    if ( ret == "") {
       ret = "NULL";
    }
    printf("Listen status = %s Errno = %s\n", ret, errno);
    print("Running...\n");
    print("\n");

    # Return the first accumulated trap in blocking mode.  Wait until a trap
    # has been received.

    res = snmp_trap_receive();
    if(res == "")
    {
        print("No SNMP trap available...errno =".errno."\n");
    }
    else
    {
        printf(res);
    }
    print("\n");

    sleep(10);

    # snmp_trap_receive() in blocking mode.  Return the next snmp trap if one exists.
    # Don’t wait.

    res = snmp_trap_receive(2);
    if(res == "")
    {
        print("No SNMP trap available...errno=".errno."\n");
    }
    else
    {
        print("SNMP Trap received:\n");
        printf(res);
    }

    # Stop accumulating SNMP traps.
    snmp_trap_ignore();
```

BMC Software, Inc., Confidential and Proprietary Information

Sample output when two SNMP traps are received. The first SNMP trap is the result of a manual SNMP trap sent using snmp_trap_send. The second SNMP trap is the result a parameter alarm:

```
Listen status = OK Errno = 0
Running...

SNMP Trap received:
172.19.201.36 1.3.6.1.4.1.1031 1 3743142
1.3.6.1.4.1.1031.1   6   string   test 1
1.3.6.1.4.1.1031.2   6   string   test 2
1.3.6.1.4.1.1031.2   6   string   test 2
1.3.6.1.4.1.1031.2   6   string   test 2

SNMP Trap received:
172.19.201.36 1.3.6.1.4.1.1031.1.1.2 5 3743929
1.3.6.1.4.1.1031.1.1.2.1.0    84  string   Alarm #2 of global parameter 'mystate'
1.3.6.1.4.1.1031.1.1.2.2.0    22  string   /PSLCOM/PSLCOM/mystate
1.3.6.1.4.1.1031.1.1.2.3.0    0   string
```

Sample output when only one SNMP trap is received. The SNMP trap is the result of a parameter alarm.

```
Listen status = OK Errno = 0
Running...

172.19.201.36 1.3.6.1.4.1.1031.1.1.2 5 3782736
1.3.6.1.4.1.1031.1.1.2.1.0    87       stringAlarm #2 of global parameter 'MEMWCache'
1.3.6.1.4.1.1031.1.1.2.2.0    24       string/MEMORY/MEMORY/MEMWCache
1.3.6.1.4.1.1031.1.1.2.3.0    0        string

No SNMP trap available...errno=0
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_trap_raise_std_trap()

Raise the standard PATROL trap

Format

`snmp_trap_raise_std_trap(`*`text`*`)`

Parameter

Parameter	Definition
text	character string that is to be included in the packet sent by the snmp_trap_raise_std_trap() function

Description

The snmp_trap_raise_std_trap() function sends the **patrolTrapV1Raised** trap with *text* in a packet to all entities registered with the **piV1mTable**. The **piV1mTable** is the table within the PATROL MIB that can be set by the external SNMP manager.

The snmp_trap_raise_std_trap() function returns the characters "OK" if successful and the NULL string otherwise.

Example

The following is an example of the snmp_trap_raise_std_trap() function:

```
opstatus = snmp_trap_raise_std_trap();
if (opstatus != "OK") {
   # raise standard trap operation failed
   # diagnosis and recovery statements here
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_trap_receive()

Return PATROL SNMP event traps

Format

```
snmp_trap_receive([flags])
```

Parameter

Parameter	Definition
flags	binary flags that control the actions of the snmp_trap_receive() function **Valid Values** 0 neither ranges 1 nor 2 applies 1 print the trap information in a formatted output 2 don't block the current process waiting for an SNMP event trap (that is, poll for the trap) 3 both flags 1 and 2 apply 4 return the full trap information including the trap specific id and community string **Default** 0

Description

The snmp_trap_receive() function returns a formatted string containing information about the first accumulated trap (if any). Only one snmp_trap_receive() function may be active on a PATROL Agent.

The snmp_trap_receive() function returns the NULL string and sets the PSL errno variable as follows:

- `E_PSL_SNMP_NOT_LISTENING` (numeric value 99) if there is no active socket established by a prior snmp_trap_listen() function

BMC Software, Inc., Confidential and Proprietary Information

- `E_PSL_SNMP_ALREADY_LISTENING` (numeric value 98) if there is already an active snmp_trap_receive() function active against the PATROL Agent

Blocking snmp_trap_receive() Function

If the snmp_trap_receive() function is specified as blocking, it blocks the current process until one of the following occurs:

- A snmp_trap_ignore() function from another PSL process ends trapping.
- The socket established by the snmp_trap_listen() function receives an SNMP trap.

In the first case, the snmp_trap_listen() function returns the NULL string. In the second case, snmp_trap_listen() unblocks the current process and returns the trap.

Nonblocking snmp_trap_receive() Function

If the snmp_trap_receive() function is specified as nonblocking, it queries the socket established by the snmp_trap_listen() function and returns any trap waiting there or the NULL string indicating that no trap was waiting.

BMC Software, Inc., Confidential and Proprietary Information

Default Trap Format

The snmp_trap_receive() function returns a trap as a character string. The format of the return string depends on flags specified:

- The default format for the string (*flags* = 0) is as follows:

  address enterprise subid uptime\n
  objectid1 type1 value1\n
  . . .
  objectidn typen valuen\n

  The *traptype* is not returned when the *flags* parameter is set to 0.

- The format for the string when *flags* = 1 or 3 is as follows:

  address enterprise traptype subid uptime\n
  objectid1 type1 value1\n
  . . .
  objectidn typen valuen\n

- The format for the string when *flags* = 4 is as follows:

  address enterprise traptype specific_id uptime community\n
  objectid1 type1 value1\n
  . . .
  objectidn typen valuen\n

Field	Definition
address	address of the SNMP agent that sent the trap
enterprise	the enterprise object identifier for the SNMP agent
traptype	SNMP trap type as defined in RFC 1213 (only returned when flags = 1 or 3) **Valid Values** 0-5 standard traps 6 enterprise trap
specific_id	the specific id of the SNMP trap
subid	identifier number for the trap

Field	Definition
uptime	number of clock ticks that the SNMP agent has been active A clock tick is hardware-dependent and for Unix systems is typically between 10 and 20 milliseconds.
community	the community string of the SNMP trap
objectidn	character string SNMP object identifier for the returned variable
typen	character string indicating the *objectidn* type Valid Values • integer • string • oid • null • counter • gauge • timeticks • ipaddress
valuen	value of *objectidn*

Each return variable consisting of an *objectidn*, *typen*, and *valuen* triplet is separated by spaces and terminated by a newline character. This format is identical to the second snmp_trap_send() function format type. Output from the snmp_trap_receive() function can be piped to the snmp_trap_send() function without modification.

Example

The following is an example of the snmp_trap_receive() function:

```
myhost = get("/hostname");
if (index(snmp_trap_register_im("","list"),myhost) == 0) {
   # host not currently registered to receive traps
   opstatus = snmp_trap_register_im(myhost,"add");
   if (opstatus != "OK") {
      # registration failed -- exit
      exit;
   }
   # at this point, it is OK to receive traps
   snmp_trap_receive(2); # 2 is non-blocking
} else {
   snmp_trap_receive(2); # 2 is still non-blocking
}
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_trap_register_im()

Maintain a list of SNMP managers that receive PATROL SNMP traps

Format

`snmp_trap_register_im(`*`host/`*`[`*`port/community`*`],[`*`action`*`])`

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
`host`	name of the computer whose SNMP manager will be added to or deleted from the list of SNMP managers that receive PATROL SNMP traps
`port`	the Universal Datagram Protocol (UDP) port number on which the SNMP agent socket opens **Default** 162
`community`	name of the character string used as a community string for SNMP operations **Default** public
`action`	character string identifying an action performed against *host/port/community* in the list of SNMP managers that receive PATROL SNMP traps **Valid Values** • **list** — return the list of SNMP managers that receive SNMP traps • **add** — add "*`host/port/community`*" to the list of SNMP managers that receive PATROL SNMP traps • **delete** — remove "*`host/port/community`*" from the list of SNMP managers that receive PATROL SNMP traps **Default** list

Description

The snmp_trap_register_im() function creates and maintains a list of SNMP V1 managers that receive PATROL SNMP traps. The list is stored in **piV1mTable** in the PATROL Management Information Base (MIB).

The snmp_trap_register_im() function returns the following:

- `OK` string if successful for an add or delete operation

- the list of registered SNMP managers for the list operation
- the NULL string if an error occurs

The snmp_trap_register_im() function can also set the PSL variable errno = 96 (E_PSL_BAD_FUNCTION_PARAMETER) upon encountering a parameter error.

Examples

These examples show how to use the snmp_trap_register_im() function.

Return the List of SNMP Managers

Each of the following commands returns the list of registered SNMP managers:

```
snmp_trap_register_im();
snmp_trap_register_im("myhost","list");
```

The list action in the second line causes the PSL interpreter to ignore the *myhost* host name.

BMC Software, Inc., Confidential and Proprietary Information

Add an SNMP Manager to the List

The following command adds the SNMP manager identified by myhost/162/public to the list of SNMP managers that receive PATROL SNMP traps:

```
snmp_trap_register_im("myhost","add");
```

The port, 162, and community, public, are defaults and do not need to be specified in the command syntax.

Delete an SNMP Manager from the List

The following command will delete the SNMP manager identified by myhost/162/defaultcommunity from the list of SNMP managers that receive PATROL SNMP traps:

```
snmp_trap_register_im("myhost/162/defaultcommunity",delete);
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_trap_send()

Send a PATROL SNMP event trap

Format

```
snmp_trap_send(host/[port/community],
      address enterprise traptype subid
      uptime,objectid1 type1 value1,...,
      [objectidn lengthn typen valuen])
```

BMC Software, Inc., Confidential and Proprietary Information

Parameters

Parameter	Definition
host	name of the computer to which the trap is sent Refer to the **/snmp/piV1m_list** description in the *PATROL Agent Reference Manual* for more information.
port	the Universal Datagram Protocol (UDP) port number to which the trap is sent on *host*
community	name of the character string used as a community string for SNMP operations on *host*
address	address of the SNMP agent that sent the trap
enterprise	the enterprise object identifier for the SNMP agent
traptype	the SNMP trap type as defined in RFC 1213 **Valid Values** 0-5 standard traps 6 enterprise trap
subid	identifier number for the trap
uptime	number of clock ticks that the SNMP agent has been active A clock tick is hardware-dependent and for Unix systems is typically between 10 and 20 milliseconds.
objectidn	character string SNMP object identifier for the included variable
lengthn	number indicating the length of *valuen*, if not delimited by \n
typen	character string indicating the *objectidn* type Valid Values • integer • string • oid • null • counter • gauge • timeticks • ipaddress
valuen	value of *objectidn*

BMC Software, Inc., Confidential and Proprietary Information

Description

The snmp_trap_send() function sends the specified trap to *host*. The snmp_trap_send() function returns the characters **OK** if it is successful and the NULL string otherwise. The snmp_trap_send() function also sets the PSL variable errno = 96 (E_PSL_BAD_FUNCTION_PARAMETER), or errno = 100 (E_PSL_SNMP_ERROR) and returns error messages to the computer icon console display window.

Example

The following is an example of the snmp_trap_send() function:

```
# Send SNMP trap to host on which PatrolAgent is running (i.e. localhost).

host = get("/hostname");
tport = get("/snmp/trap_port");
arg1 = host."/".tport;
arg2 = host." .1.3.6.1.4.1.1031 6 1 1";
print(snmp_trap_send(arg1,arg2,
          ".1.3.6.1.4.1.1031.1 string test",
          ".1.3.6.1.4.1.1031.2 string test 2",
          ".1.3.6.1.4.1.1031.3 string test 3",
          ".1.3.6.1.4.1.1031.4 string test 4"));
```

BMC Software, Inc., Confidential and Proprietary Information

snmp_walk()

Return subsequent variables from the Management Information Base (MIB) on the agent that the SNMP session is opened with

Format

`snmp_walk(`*`session`*`,`*`variable`*`)`

Parameters

Parameter	Definition
session	ID of the SNMP session to which the series of SNMP get-next-request commands will be submitted The *session* is the ID returned when an snmp_open() function is executed.
variable	name of an SNMP MIB variable, which can be specified in numeric form or in symbolic form if the MIB is known to the PATROL Agent

Description

The snmp_walk() function returns all variables in the subtree for *variable* in the Management Information Base (MIB). The subtree for *variable* includes all variables of the form *variable.argument_sequence* where *variable* is the prefix name.

The snmp_walk() function returns the NULL string if an error occurs and sets the PSL errno variable as follows:

- E_PSL_TIMEOUT if a timeout occurs during the request
- E_PSL_SNMP_ERROR if an error is returned in the response packet from the SNMP session

BMC Software, Inc., Confidential and Proprietary Information

Output Format

The output of the snmp_walk() function is a s list of SNMP variables in the following format:

name type length value

Field	Definition
name	name of an SNMP MIB variable
type	character string indicating the *name* type
length	length of *value*
value	value of *name*

The timeticks, gauge, and counter variable types are followed by an extra new line character (\n). The extra new line character creates a blank line after these variable types in the snmp_walk() function output.

Example

The following example uses the snmp_walk() function to print all the application instances in the PATROL MIB. It is a more efficient form of a similar example using the snmp_get_next() function (see page 1-446).

```
function main() {
   local myhost,session,app_start;
   myhost = get("/hostname");
   session = snmp_open(myhost,"1","","","",8161);
   app_start = ".1.3.6.1.4.1.1031.1.1.1.7.1.1";
   printf("%s\n",snmp_walk(session,app_start));
   snmp_close(session);
   return;
}
```

sort()

Sort a list of numeric or alphabetic values

Format

`sort(`*`list`*`,[`*`mode`*`],[`*`position`*`])`

Parameters

Parameter	Definition
`list`	a PSL list whose elements that are to be sorted
`mode`	optional character string specifying the sort order **Valid Values** n ascending numeric order nr descending numeric order "" ascending alphabetic order r descending alphabetic order **Default** `""` (ascending alphabetic)
`position`	optional integer that specifies the character position within each element of *list* where sorting is to begin The first character of each list element is character 1. If the length of every element within list is less than *position*, the effect is the same as if all the elements of list were NULL elements. The *position* does not truncate elements; it only ignores the first (*position* – 1) characters for purposes of comparison. **Default** 1

BMC Software, Inc., Confidential and Proprietary Information

Description

The sort() function returns a sorted version of *list* that is ordered according to *mode*.

The sort() function does not merge duplicate entries in *list*. The returned list has the same number of members as *list*. The order in which duplicates are returned is not defined because it is not defined for the C library qsort() function. This fact is relevant for the following cases:

- numeric sorting of strings with identical numeric prefix values but different nonnumeric suffixes
- any sorting mode in which *position* is larger than more than one element within *list* (the sort() function regards all such elements as duplicate NULL elements)

If *list* is the NULL list, the sort function returns the NULL list. For a nonempty *list*, the sort() function always returns a well-defined list with the last line properly terminated by a new-line character.

Note

The *list* does not need to be terminated by a new-line character. Numeric sorting is based on floating point values; nonnumeric list entries are converted according to the system's standard C library function atof().

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example shows the different sorting orders available in the sort() function:

```
function print_list(label,array) {
   printf("%s",label);
   foreach member (array) {
      printf("%2d ",trim(member,"\n"));
   }
   printf("\n");
}

function main() {
   srandom(time());
   i = 0;
   while (i <= 15) {
      list = [list,random(100)];
      i++;
   }
   print_list("unsorted list        : ",list);
   print_list("sorted asc. numeric  : ",sort(list,"n"));
   print_list("sorted desc. numeric : ",sort(list,"nr"));
   print_list("sorted asc. alpha    : ",sort(list,""));
   print_list("sorted desc. alpha   : ",sort(list,"r"));
}
```

The example produces the following output:

```
unsorted list        : 90  6 68 84 59 52  4 67 74 15 48 46 34 88 76 27
sorted asc. numeric  :  4  6 15 27 34 46 48 52 59 67 68 74 76 84 88 90
sorted desc. numeric : 90 88 84 76 74 68 67 59 52 48 46 34 27 15  6  4
sorted asc. alpha    : 15 27 34  4 46 48 52 59  6 67 68 74 76 84 88 90
sorted desc. alpha   : 90 88 84 76 74 68 67  6 59 52 48 46  4 34 27 15
```

BMC Software, Inc., Confidential and Proprietary Information

splitline()

Parse text into user-defined variables

Format

```
splitline(text, delimiters, flag, variable_1,
   variable_2,[...variable_n])
```

Parameter

Parameter	Definition
text	text that you want to parse
delimiter	character(s) that separates the fields
flag	one or more literal characters that affect the behavior of this function **Valid Values** " " returns the number of fields placed into variables t returns the total number of fields that can be extracted a if the text can be parsed into more fields than the number of variables, the last variable contains all of the remaining text f consecutive occurrences of the delimiter are interpreted as individual fields (see description for more information)
variable_1, [variable_2, ...variable_n]	one or more user-defined variables to hold the extracted fields You can use a maximum of 97 variables. Define and initialize variables before using them with splitline(). This function ignores any literal strings passed as variables.

BMC Software, Inc., Confidential and Proprietary Information

Description

The splitline() function parses text into fields (based on delimiters) and assigns these fields to user-supplied variables. This function returns the number of fields saved to variables. See the *flags* parameter in the preceding table for more information about return values.

The default behavior of splitline() is to interpret two or more consecutive delimiters as being a single delimiter. With the f flag, splitline() reads every occurrence of a delimiter as being the end of a field. For example, splitline() typically reads the following string as having three fields, "abc::123:xyz" (where ':' is the delimiter). With the f flag, splitline() reads the same string as having four fields, and the second field contains no data. Using the f flag with splitline() is similar to using the nthargf() function.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example demonstrates the splitline() function:

```
Data = cat("/etc/passwd");
foreach line x (Data) {
   splitline(x, ":", "", User, Passwd, Uid);
   printf("%s %d\n", User, Uid);
}
```

The following example shows the use of the f flag:

```
splitline("abc::123", ":", "", var_1, var_2, var_3);
print("Default:var 1=". var_1.", var 2=". var_2.",
   var 3=". var_3);
splitline("abc::123", ":", "f", var_1, var_2, var_3);
print("\nf flag:var 1=". var_1.", var 2=". var_2.",
   var 3=". var_3);
```

The following output was generated by the previous example:

```
Default:var 1=abc, var 2=123, var 3=
f flag:var 1=abc, var 2=, var 3=123
```

BMC Software, Inc., Confidential and Proprietary Information

sprintf()

Return the specified format as a character string to a destination

Format

`sprintf(`*`format`*`)`

Parameter

Parameter	Definition
format	text, variable names, and control characters that specify the content and format of the character string output to the computer or task output window The *format* permits the following conversion characters: %d signed decimal (identical to %i) %i signed decimal (identical to %d) %u unsigned decimal %o unsigned octal %x unsigned hexadecimal using `abcedf` %X unsigned decimal using `ABCDEF` %c unsigned character %s character string %e double-precision form `drddde±ddd` where each `d` is a digit and `r` is the radix character %E double-precision form `drdddE±ddd` where each `d` is a digit and `r` is the radix character %f decimal notation form `dddrddd` where each `d` is a digit and `r` is the radix character %g print in the style of %e if the exponent after conversion is less than –4, else print in style %f %G print in the style of %E with the precision specifying the number of significant digits %N group digits into threes and separate with commas beginning at the right of the string %% print a % character The *format* does not support he standard C pointer conversion characters %p and %n.
`format` (continued)	The *format* permits the following flags: – left-justify and pad on the right with spaces + display plus sign when value is greater than zero 0 pad with zeros if no other padding is specified # alters the meaning of a conversion: •appends 0x or 0X to the %x and %X conversions •always appends the radix character to the %e, %E, %f, %g, and %G conversions •retains trailing zeros in the %g and %G conversions The # flag does not affect the %c, %d, %s, or %i conversions.

BMC Software, Inc., Confidential and Proprietary Information

Description

The sprintf() function is identical to the printf() function except that it returns the created string rather than outputting it to the PATROL Console. If there is an error in format, sprintf sets the PSL errno variable and returns the NULL string.

The formats, conversions, and values of errno for the various errors are identical to those described for the printf() function.

C programmers should be careful to use the PSL style:

```
destination=sprintf(format)
```

rather than the C-style:

```
sprintf(destination,format)
```

The latter style will often cause a compilation warning about a null-effect statement.

C Conventions Not Supported by the PSL sprintf Function

The sprintf() function does not support the C convention of using the asterisk (*) as a field width or precision indicator. The sprintf() function does not support the `%p` and `%n` conversion characters.

The length modifiers `h`, `l` (ell), and `L` are not valid and are ignored by the sprintf() function.

The sprintf() function format conversions are passed directly to the C library sprintf() routine on each platform. The output for obscure formatting features may differ across platforms, especially on platforms with libraries that do not conform to ANSI C requirements.

Conversion Differences between the C sprintf Routine and PSL sprintf Function

The format conversions have the same meaning between standard C and PSL, but the concept of variable types differs between the two.

BMC Software, Inc., Confidential and Proprietary Information

PSL supports only string types for its variables, and thus string arguments to the sprintf() function are converted in a manner appropriate for the format conversion:

- Integral formats, such as `%d`, convert the string to signed integers.
- Noninteger numeric formats, such as `%f`, convert to floating point values.
- `%c` prints the ASCII equivalent of its integer argument, or for nonnumeric arguments, the first character of its argument. (Applying `%c` to “65” will print ‘A’ and to “AB” will print ‘A’.)
- `%s` causes no conversion.
- `%%` requires no argument.

The %N Format Conversion

The sprintf() function provides one nonstandard C extension—the `%N` conversion. The `%N` conversion preprocesses a numeric string so that commas separate each group of three digits beginning at the right side of the string.

For example, the `%N` conversion causes the following conversions:

1234 ⇒ 1,234
12345 ⇒ 12,345
123456 ⇒ 123,456

The `%N` conversions ignores initial minus signs and blanks while searching for the first sequence of digits so that `%N` can be applied to negative values. If no digits are found after the skipped characters, the printed argument is unchanged.

The `%N` conversion only modifies the first sequence of digits. For example, the `%N` conversion changes floating point numbers like 1234.1234 to become 1,234.1234 without changing to the digit sequence to the right of the decimal point.

BMC Software, Inc., Confidential and Proprietary Information

As part of the `%N` conversion, the sprintf() function performs a %s conversion using the field width and precision specifiers supplied in *format*. The sprintf() function prints the string resulting from the combined `%N` and `%s` conversions. Because of the embedded `%s` conversion, field width and precision under `%N` conversion have the same effect as with `%s`.

Note

Currently, no localization is supported by `%N`, and so the formatting achieved by `%N` does not change in different locales.

Example

The following PSL script uses the sprintf() statement to recursively build a formatted decimal/octal/hexadecimal conversion table before printing it.

```
function main() {
   i = 0;
   string = sprintf(" n    Oct   Hex\n");
   string = sprintf("%s---  ----  ----\n",string);
   while (i++ <= 25) {
      string = sprintf("%s%3d  %4o  %4X\n",string,i,i,i);
   }
   print(string);
   return;
}
```

The example produces the following output:

```
 n    Oct   Hex
---  ----  ----
  1     1     1
  2     2     2
  3     3     3
  4     4     4
  5     5     5
  6     6     6
  7     7     7
  8    10     8
  9    11     9
```

BMC Software, Inc., Confidential and Proprietary Information

```
10    12     A
11    13     B
12    14     C
13    15     D
14    16     E
15    17     F
16    20    10
17    21    11
18    22    12
19    23    13
20    24    14
21    25    15
22    26    16
23    27    17
24    30    18
25    31    19
```

BMC Software, Inc., Confidential and Proprietary Information

sqrt()

Return the square root of the argument

Format

`sqrt(`*`argument`*`)`

Parameter

Parameter	Definition
argument	numeric value whose mathematical square root is returned **Valid Values** $0 \leq argument < \infty$

Description

The sqrt() function returns the square root of the positive integer or real value *argument*.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL script compares the output of the sqrt() function with that derived from the trigonometric identity $\ln(x^a) = a \ln(x)$ for the integers 1 through 10:

```
function main() {
   printf("SQUARE ROOT AND EXP(0.5 * LOGE) COMPARISON\n\n");
   printf("    n  sqrt(n)  exp(0.5 * loge(n))\n");
   printf("   --  -------  -----------------\n");
   local i;
   i = 0;
   while (i++ <= 10) {
      printf("   %2d  %7.5f       %7.5f\n",i,sqrt(i),exp(0.5 * loge(i)));
   }
}
```

The example produces the following output:

```
SQUARE ROOT AND EXP(0.5 * LOGE) COMPARISON
    n  sqrt(n)  exp(0.5 * loge(n))
   --  -------  -----------------
    1  1.00000       1.00000
    2  1.41421       1.41421
    3  1.73205       1.73205
    4  2.00000       2.00000
    5  2.23607       2.23607
    6  2.44949       2.44949
    7  2.64575       2.64575
    8  2.82843       2.82843
    9  3.00000       3.00000
   10  3.16228       3.16228
```

BMC Software, Inc., Confidential and Proprietary Information

srandom()

Initialize the random number generator with a seed value

Format

`srandom(`*`seed`*`)`

Parameter

Parameter	Definition
seed	numeric value used as a starting point for pseudorandom number generation by the random() function

Description

The srandom() function sets the random number seed for the `random()` function. The *seed* is passed directly to the Unix C srandom() function.

The PSL srandom() function always returns the NULL string.

Example

The following is an example of the srandom() function:

```
srandom(time());
i = 0;
while (i++ <= 5) {
   printf("random number %1d is: %3d\n",i,random(100));
}
```

BMC Software, Inc., Confidential and Proprietary Information

subset()

Verify that one PSL list is a subset of another

Format

`subset(`*`set`*`,`*`subset`*`)`

Parameters

Parameter	Definition
set	a PSL list that is the set in the set-subset verification
`subset`	a PSL list that is the subset in the set-subset verification

Description

The subset() function returns a Boolean value of 0 or 1 indicating whether *subset* is a proper or improper subset of *set*. If *subset* is the NULL set, the subset() function returns 1 (TRUE). If *set* is the NULL set and *subset* is not, the subset() function returns 0 (FALSE).

The subset() function ignores duplicates and returns 1 only if all elements of *subset* are also present in *set*.

Example

The subset() function can be used to determine whether a particular element is present in a set and thus provides “is_member” functionality such as the following:

```
if(subset(my_set,"blue"))
{
   # PSL set "my_set" contains element "blue"
}
```

It is not necessary to place a new line at the end of the “blue” string because it is inserted by the subset() function. The example statements are treated as a subset() function acting on a set with one element.

BMC Software, Inc., Confidential and Proprietary Information

substr()

Return a specified portion of a string of characters

Format

`substr(`*`text`*`,`*`start`*`,`*`length`*`)`

Parameters

Parameter	Definition
`text`	text from which a substring is to be returned The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
`start`	the character position within *text* that is to be the first character of the substring The first character in *text* is character position 1.
`length`	the total number of characters returned from *text*

Description

The substr() function returns the substring of *text* of *length* characters that starts at position *start*.

Example

The following is an example of the substr() function:

```
any_string = "The quick red fox jumped over the lazy brown dogs";
len - length(any_string);
i = 0;
while (i++ <= len) {
   printf("%s\n",substr(any_string,i,1));
}
```

BMC Software, Inc., Confidential and Proprietary Information

system()

Submit a command to the computer operating system

Format

`system(`*`command`*`,[`*`instance`*`])`

Parameters

Parameter	Definition
`command`	syntax of the submitted operating system command The *command* parameter can contain output redirection, pipes, wild cards, and so on.
`instance`	optional application instance against which *command* executes **Default** The application instance that is the nearest ancestor of *command.*

Description

The system() function returns any output to stdout/stderr produced by submitting *command* to the operating system. Unlike the execute() function, *command* is not executed directly but rather at the Unix command line:

`sh -c` *`command`*

Note

The shell used to execute *command* is the user login shell of the account where the system() function executes. For OS/2, the shell used to execute *command* is the shell specified in the COMSPEC environment variable.

BMC Software, Inc., Confidential and Proprietary Information

Sometimes it is desirable to submit long-running commands like daemons to run in the background. For more information on submitting commands to run in the background see the popen() function on page 1-265.

Example

The following is an example of the system() function:

```
if (get("/appType") == "NT") {
   printf("%s\n",system("net user"));
} else {
   printf("%s\n",system("cat /etc/passwd"));
}
```

tail()

Return the last lines from a text block

Format

`tail(`*`text`*`,`*`lines`*`)`

Parameters

Parameter	Definition
`text`	text whose last lines are to be returned The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
`lines`	number of lines of *text* to be returned, starting from the last line of *text*

Description

The tail() function returns the last *lines* number of lines of *text*. It is equivalent to the Unix tail(1) command.

Note

The tail() function does *not* perform file I/O. If reading *text* from a file, you must use the cat() function for file I/O. For more information about the cat() function, see page 1-38.

Example

The following example uses the tail() function to return the last three lines from a PSL list:

```
psl_list =
["line01","line02","line03","line04","line05","line06","line07","line08"];
printf("%s\n",tail(psl_list,3));
```

The example produces the following output:

```
line06
line07
line08
```

BMC Software, Inc., Confidential and Proprietary Information

tan()

Return the tangent of the argument

Format

`tan(`*`radians`*`)`

Parameter

Parameter	Definition
radians	arc length in radians whose tangent is to be determined **Valid Values** $-\infty \leq radians \leq \infty$

Description

The tan() function returns the tangent of *radians*. The output range for the `tan()` function is $-\infty < \tan() < \infty$. The tan() function is undefined when radians = $\pi(2n+1)/2$ where n is an integer.

Example

The following example compares the output of the tan() function with the output of its trigonometric definition sin(x)/cos(x):

```
PI = 3.141593;
   printf("   degrees   tan(x)   sin(x)/cos(x)\n");
   printf("   -------  --------  -------------\n");
   i = -10;
   while ((i += 10) <= 360) {
   radians = i * 2 * PI / 360;
      if (fabs(cosine = cos(radians)) <= 0.000001) {
         printf("     %3d*** tan(x) is undefined for this value\n",i);
         next;
      }
      printf("     %3d    %+8.5f    %+8.5f\n",i,tan(radians),sin(radians) / cosine);
   }
```

The example produces the following output:

```
degrees   tan(x)   sin(x)/cos(x)
   -------  --------  -------------
       0    +0.00000    +0.00000
      10    +0.17633    +0.17633
      20    +0.36397    +0.36397
      30    +0.57735    +0.57735
      40    +0.83910    +0.83910
      50    +1.19175    +1.19176
      60    +1.73205    +1.73205
      70    +2.74748    +2.74748
      80    +5.67130    +5.67129
      90*** tan(x) is undefined for this value
     100    -5.67129    -5.67129
     110    -2.74748    -2.74748
     120    -1.73205    -1.73205
     130    -1.19175    -1.19175
     140    -0.83910    -0.83910
     150    -0.57735    -0.57735
     160    -0.36397    -0.36397
     170    -0.17633    -0.17633
     180    +0.00000    +0.00000
     190    +0.17633    +0.17633
     200    +0.36397    +0.36397
     210    +0.57735    +0.57735
     220    +0.83910    +0.83910
     230    +1.19175    +1.19176
     240    +1.73205    +1.73206
     250    +2.74749    +2.74749
     260    +5.67131    +5.67132
     270*** tan(x) is undefined for this value
     280    -5.67128    -5.67129
     290    -2.74748    -2.74748
     300    -1.73205    -1.73205
     310    -1.19175    -1.19175
     320    -0.83910    -0.83910
     330    -0.57735    -0.57735
     340    -0.36397    -0.36397
     350    -0.17633    -0.17633
     360    +0.00000    +0.00000
```

BMC Software, Inc., Confidential and Proprietary Information

tanh()

Return the hyperbolic tangent of the argument

Format

`tanh(`*`argument`*`)`

Parameter

Parameter	Definition
argument	numeric value whose hyperbolic tangent is to be determined **Valid Values** $-\infty \leq$ *argument* $\leq \infty$

Description

The tanh() function returns the hyperbolic tangent of *argument*.The hyperbolic tangent is defined by the expression:

$$\tanh(x) = (e^{x} - e^{-x})/(e^{x} + e^{-x})$$

where e is the base for the natural logarithms (e = 2.71828 . . .). The output range for the tanh() function is $-1 \leq \tanh() \leq 1$.

Example

The following example verifies the output of the tanh() function against the output of its definition in the range $-2 \leq x \leq 2$:

```
function main() {
    printf("HYPERBOLIC TANGENT TEST\n\n");
    printf("     x     tan(x)   definition\n");
    printf("   -----  --------  ----------\n");
    i = -2.1;
    while ((i += 0.1) <= 2.0) {
       eplusx = exp(i);
       eminusx = exp (-i);
       printf("   %+5.2f  %+8.5f   %+8.5f\n",i,tanh(i),(eplusx - eminusx) / (eplusx + eminusx));
    }
}
```

The example produces the following output:

```
HYPERBOLIC TANGENT TEST
     x     tan(x)   definition
   -----  --------  ----------
   -2.00  -0.96403   -0.96403
   -1.90  -0.95624   -0.95624
   -1.80  -0.94681   -0.94681
   -1.70  -0.93541   -0.93541
   -1.60  -0.92167   -0.92167
   -1.50  -0.90515   -0.90515
   -1.40  -0.88535   -0.88535
   -1.30  -0.86172   -0.86172
   -1.20  -0.83366   -0.83366
   -1.10  -0.80050   -0.80050
   -1.00  -0.76159   -0.76159
   -0.90  -0.71630   -0.71630
   -0.80  -0.66404   -0.66404
   -0.70  -0.60437   -0.60437
   -0.60  -0.53705   -0.53705
                              .
                              .
                              .
                              .
```

BMC Software, Inc., Confidential and Proprietary Information

text_domain()

Set or get the name of the current domain

Format

`text_domain(`*`domain_name`*`)`

Parameter

Parameter	Definition
domain_name	file name for message catalog Use only characters that are permitted in file names. **Valid Values** 0, "", or any valid domain name **Default** messages

Description

The text_domain() function sets or gets the name of the current domain. The *domain_name* determines the name of a string-based catalog file according to the following format:

PATROL_HOME/**lib/nls/locale**/*locale_name*/**ML**_*category*/*domain_name*.**bo**

BMC Software, Inc., Confidential and Proprietary Information

Using 0 as *domain_name* causes text_domain() to return the value of the current domain name. When *domain_name* is the NULL character ("), text_domain() sets the current domain to the default value (messages). Using a valid domain name for *domain_name* sets this value as the domain. When text_domain() successfully sets the domain, it returns the new value of the domain. You may want to use text_domain() before get_text() because get_text() uses the current setting of the domain name.

Note

This function provides support for internationalized PSL scripts. For more information about internationalization, see *PATROL Script Language Reference Manual Volume 1 —PSL Essentials*.

Example

The following example demonstrates the text_domain() function.

```
ret = text_domain(0);
printf("Current domain is %s\n", ret);

ret = text_domain("test_msg");
printf("Domain set to %s \n", ret);

ret = text_domain("");
printf("Domain set to default value of %s \n", ret);
```

The previous example produces the following output:

```
Current domain is messages
Domain set to test_msg
Domain set to default value of messages
```

BMC Software, Inc., Confidential and Proprietary Information

time()

Return the number of seconds since 00:00:00 GMT January 1, 1970

Format

```
time()
```

Description

The time() function returns the current local system time as the number of seconds that have elapsed since 00:00:00 GMT, Jan 01, 1970.

Example

The following is an example of the time() function:

```
function main() {
   local seconds,current;
   seconds = time();
   current = asctime(seconds);
   printf("seconds since 00:00:00 GMT January 1, 1970 : %d\n",seconds);
   printf("     current date and time using asctime() : %s\n",current);
   printf("        current date and time using date() : %s\n",date());
   printf("              seconds using convert_date() : %d\n",convert_date(current));
}
```

The example produces the following output:

```
seconds since 00:00:00 GMT January 1, 1970 : 877458704
     current date and time using asctime() : Tue Oct 21 13:31:44 1997
        current date and time using date() : Tue Oct 21 13:31:44 1997
              seconds using convert_date() : 877458704
```

BMC Software, Inc., Confidential and Proprietary Information

tmpnam()

Return a unique name for temporary file creation

Format

```
tmpnam()
```

Description

The tmpnam() function returns a name that is guaranteed to be unique and can be used to pass to the fopen function for creating temporary files.

The semantics of the tmpnam() function are similar to that of the C tmpnam() routine—notably, a restricted number of unique names are returned by the tmpnam() routine as defined by the C constant TMP_MAX. All PSL processes on a given PATROL Agent share the same set of names, and there can be a danger of mixing names. ANSI C guarantees that TMP_MAX is at least 25, but it is typically much larger than that. If the size of TMP_MAX is a concern, add a suffix to the returned file name.

Example

The following example shows how to use the tmpnam() function to generate a temporary file name. The PSL function adds a suffix to the returned name to further guarantee its uniqueness.

```
name = tmpnam() . ".dave";
fp = fopen(name, "w");
```

BMC Software, Inc., Confidential and Proprietary Information

tolower()

Convert text to all lowercase characters

Format

`tolower(`*`text`*`)`

Parameter

Parameter	Definition
text	text that is to be returned as lowercase letters The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.

Description

The tolower() function returns a copy of *text* with all uppercase letters converted to lowercase letters.

Example

The following is an example of the tolower() function:

```
function main() {
   teststr = "SOME ARE UPPER, some are lower, SoMe ArE mIxEd";
   printf(" original : %s\n",teststr);
   printf("all lower : %s\n",tolower(teststr));
}
```

The example produces the following output:

```
 original : SOME ARE UPPER, some are lower, SoMe ArE mIxEd
all lower : some are upper, some are lower, some are mixed
```

BMC Software, Inc., Confidential and Proprietary Information

toupper()

Convert text to all uppercase characters

Format

`toupper(`*`text`*`)`

Parameter

Parameter	Definition
text	text that is to be returned as uppercase letters The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.

Description

The toupper() function returns a copy of *text* with all lowercase letters converted to uppercase letters.

Example

The following is an example of the toupper() function:

```
function example_20() {
   teststr = "SOME ARE UPPER, some are lower, SoMe ArE mIxEd";
   printf(" original : %s\n",teststr);
   printf("all upper : %s\n",toupper(teststr));
}
```

The example produces the following output:

```
 original : SOME ARE UPPER, some are lower, SoMe ArE mIxEd
all upper : SOME ARE UPPER, SOME ARE LOWER, SOME ARE MIXED
```

BMC Software, Inc., Confidential and Proprietary Information

trace_psl_process()

Enable PSL tracing for a specified application, parameter, or process

Format

`trace_psl_process(`*`what`*`,`*`name`*`,`*`value`*`)`

Parameters

Parameter	Definition
`what`	the PSL component of a KM that should have tracing enabled This determines how the PTROL Agent will find *name*. Legal values are DISCOVERY, PREDISCOVERY, PARAMETER, and PID. Case is insignificant, and unique prefixes are allowed.
`name`	the name of the application or parameter, or the PSL pid of the process
`value`	the value to be set to the process (identified by *what* and *name*) *PslDebug* variable **Valid Values** any value that *PslDebug* can be set to

Description

Use the trace_psl_process() function to enable PSL tracing for a specified application, parameter, or process. In previous releases, setting the *PslDebug* variable in the script was the only way to trace processes.

The trace_psl_process() function provides a method to set the *PslDebug* variable for a particular process without having to put the assignment in the script text. The variable *PslDebug* is persistent across executions and is not reset when the process reinitializes.

BMC Software, Inc., Confidential and Proprietary Information

The function returns 1 if the call was successful and 0 if it was not. Usually, a return value of 0 means that the process could not be found.

Examples

The following examples show different settings for the *PslDebug* variable:

```
# Enable all tracing for the discovery process of the foo application
status = trace_psl_process("DISC","foo",-1);

# Disable all tracing for the pre-discovery process of the foo application
status = trace_psl_process("PRE-DISC","foo",0);

# Enable function argument tracing in foo’s foobar parameter for the
# bar instance
status = trace_psl_process("PARAM","/foo/bar/foobar",1024);

# Enable variable assignment tracing and function argument tracing in
# foo’s foobar parameter of the bar instance
status = trace_psl_process("PARAM","/foo/bar/foobar",1024|4096);

# Enable all tracing of the PSL process with the pid of 42
status = trace_psl_process("PID",42,-1);
```

BMC Software, Inc., Confidential and Proprietary Information

trim()

Remove unwanted characters from text

Format

```
trim(str,chars,[behavior])
```

Parameters

Parameter	Definition
str	string to be returned without specified characters The *str* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.
chars	one or more characters that are to be removed from the copy of *str* output by the trim() function
behavior	optional argument that defines the behavior of the PSL trim() function The value of *behavior* is specified as either an integer or one of the following constants: 0 TRIM_ALL–trims all occurrences of *chars* from *str* (PSL 3.3 behavior) 1 TRIM_LEADING–trims *chars* from the front of the string, stopping when it encounters a character that is not specified in *chars* 2 TRIM_TRAILING– only trims *chars* from the end of the string 3 TRIM_LEADING_AND_TRAILING– trims *chars* from the start and the end of the string 4 TRIM_REDUNDANT–trims redundant *chars* from the string when they are adjacent to each other. When redundant characters are removed one of the characters is left in the string. So if you had three characters in a row, you would have one after the trim() function removed redundant characters. **Default** 0 (TRIM_ALL)

BMC Software, Inc., Confidential and Proprietary Information

Description

The trim() function returns a copy of *str* with specified characters removed. The trim() function allows you to remove specified characters from the entire string, the beginning of the string, the end of the string, or both the beginning and end of the string. The trim() function will also remove redundant characters in the string.

Example

The following example demonstrates the trim() function by manipulating a string using all behaviors of the trim() function:

```
$include "stdpsl.ph"
using stdpsl;
void_t main() {
string_t string, trimmed_string;
string= "LEADING\nXX\nTh$is \nlo^oks \nbet!ter \nwith \nunw*anted
\nchar##acters \nremo@ved..\nXX\nTRAILING";
trimmed_string = trim(string,"\n$^!*#@");
print(string,"\n");
print("\n",trimmed_string);
trimmed_string = trim(trimmed_string,"LEADING",TRIM_LEADING);
print("\n",trimmed_string);
trimmed_string = trim(trimmed_string,"TRAILING",TRIM_TRAILING);
print("\n",trimmed_string);
trimmed_string = trim(trimmed_string,"XX",TRIM_LEADING_AND_TRAILING);
print("\n",trimmed_string);
trimmed_string = trim(trimmed_string,".",TRIM_REDUNDANT);
print("\n",trimmed_string);
}
```

BMC Software, Inc., Confidential and Proprietary Information

This example produces the following output:

```
LEADING
XX
Th$is
lo^oks
bet!ter
with
unw*anted
char##acters
remo@ved..
XX
TRAILING

LEADINGXXThis looks better with unwanted characters removed..XXTRAILING
XXThis looks better with unwanted characters removed..XXTRAILING
XXThis looks better with unwanted characters removed..XX
This looks better with unwanted characters removed..
This looks better with unwanted characters removed.
```

union()

Return a list that is the union of individual lists

Format

`union(`*`list1,list2,`*`[`*`list3,list4...listn`*`])`

Parameters

Parameter	Definition
listn	the PSL list containing elements that shall be united and returned in a single well-defined list Only the first two input lists, *list1* and *list2*, are required; all others are optional.

Description

The union() function returns a PSL list that contains the elements from all *listn* merged together. Unlike the difference() and intersection() functions, the list returned by the union function is a well-defined set without any duplicates.

The union(*x*,*y*) function is similar to specifying the unique(*x*,*y*) function using the string concatenation operator "." to perform union() and the unique() function to remove duplicates. The advantage in using the union() function over the unique() function occurs when x has no terminating new-line character, in which case the use of the "." operator could cause the last element of *x* and the first element of *y* to become treated as a single set element. The union() function adds a new line to the end of every nonempty list that is missing one.

If the return value is not the NULL list, the returned set is always terminated by a new line so that all set elements end with a new-line character.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the union() function:

```
function main() {
   local group1,group2,group3,group4;
   group1 = [1,2,3];
   group2 = [3,4,5];
   group3 = [5,6,7];
   group4 = [7,8,9,10];
   printf("%s\n",union(group1,group2,group3,group4));
   # or you can do it this way:
   printf("%s\n",union(group1,union(group2,union(group3,group4))));
   return;
}
```

Each of the two printf() function statements prints a PSL list containing the integers 1 through 10.

BMC Software, Inc., Confidential and Proprietary Information

unique()

Remove the duplicate elements from a list

Format

`unique(`*`list`*`)`

Parameter

Parameter	Definition
list	a PSL list containing elements that shall be returned in a single well-defined (unique) list

Description

The unique() function returns a well-defined PSL list with all duplicates removed. All elements that remain in the return value appear in the same order as they did in *list*. If *list* is the NULL list, the unique() function returns the NULL list; otherwise the unique() function returns a list that is terminated by a new-line character so that all list elements in the list end with a new-line character.

Example

The following is an example of the unique() function:

```
function main() {
   local list;
   list = [1,2,3,3,4,5,5,6,6,6,7,8,7,9,7,10];
   printf("%s\n",unique(list));
   return;
}
```

The printf() function statement prints a PSL list containing the integers 1 through 10.

unlock()

Release a PSL process lock

Format

`unlock(`*`lockname`*`)`

Parameter

Parameter	Definition
lockname	name of the lock that should be released

Description

The unlock() function releases *lockname* that was granted to this process by a previous call to the lock() function. The unlock() function returns 1 for success and 0 for failure. If no lock is named *lockname* or if it is not currently owned by this process, then the unlock() function reports a run-time error, sets the PSL errno variable, and returns 0.

Note

All locks held by a process are automatically released when the process exits in a manner similar to executing the unlock function. BMC Software recommends that you release locks explicitly using the unlock() function rather than implicitly using the process exit.

If this process is the only one holding the lock and processes are queued for it, the first waiting process is awakened and granted use of the lock. If the first process is a shared request, then any other processes that are queued for a shared lock are also granted shared access to the lock (except for processes that are behind a writer request on the queue for this lock).

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the unlock() function:

```
if (unlock(PASSWD_LOCK)) {
   printf("lock released\n”);
} else {
   printf("lock release failed\n”);
}
```

BMC Software, Inc., Confidential and Proprietary Information

unset()

Remove a PATROL object that was previously assigned by the `set()` function

Format

`unset(`*`object`*`)`

Parameters

Parameter	Definition
`object`	text string name of the PATROL object that is to be removed by the unset() function The *object* must have been created by the set() function and the caller must have write permission to *object*.

Description

The unset() function removes the value of *object* that was previously assigned by the set() function. The unset() function always returns the NULL string and will set the PSL variable errno = 96 (E_PSL_BAD_FUNCTION_PARAMETER) if *object* is not found or if the caller does not have read permission to *object*.

BMC Software, Inc., Confidential and Proprietary Information

va_arg()

Retrieve the next argument in a function's variable argument list

Note

The va_arg() function is used only within user-defined function() statements that specify a variable list of arguments in the function() definition.

Format

`va_arg(`*`va-list`*`)`

Parameters

Parameter	Definition
`va-list`	a PSL list containing a variable number of arguments

Description

The va_arg() function returns the next argument from the variable-length list *va-list*. If there are no more arguments in *va-list*, the va_arg() function returns the NULL string and sets the PSL variable errno = 69 (E_PSL_ERANGE).

When used with the va_start() function, the va_arg() can read a variable argument list multiple times, using the va_start() function to reset the list after each reading. (The va_start() function always resets the variable argument list to return the first argument.)

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the va_arg() function:

```
function summation(...) {
   c = sum = 0;
   terms = va_start();
   c = va_arg(terms);
   while (c != "") {
      sum += c;
      c = va_arg(terms);
   }
   return sum;
}

function main() {
   print("Sum of the first 2 numbers is : ",summation(1,2),"\n");
   print("Sum of the first 3 numbers is : ",summation(1,2,3),"\n");
   print("Sum of the first 4 numbers is : ",summation(1,2,3,4),"\n");
   print("Sum of the first 5 numbers is : ",summation(1,2,3,4,5),"\n");
   print("Sum of the first 6 numbers is : ",summation(1,2,3,4,5,6),"\n");
   print("Sum of the first 7 numbers is : ",summation(1,2,3,4,5,6,7),"\n");
   print("Sum of the first 8 numbers is : ",summation(1,2,3,4,5,6,7,8),"\n");
   print("Sum of the first 9 numbers is : ",summation(1,2,3,4,5,6,7,8,9),"\n");
   return;
}
```

The example produces the following output:

```
Sum of the first 2 numbers is : 3.000000
Sum of the first 3 numbers is : 6.000000
Sum of the first 4 numbers is : 10.000000
Sum of the first 5 numbers is : 15.000000
Sum of the first 6 numbers is : 21.000000
Sum of the first 7 numbers is : 28.000000
Sum of the first 8 numbers is : 36.000000
Sum of the first 9 numbers is : 45.000000
```

BMC Software, Inc., Confidential and Proprietary Information

va_start()

Reset a function's variable argument list to return the first argument

Note

The va_start() function is used only within user-defined function() statements that specify a variable list of arguments in the function() definition.

Syntax

```
va_start()
```

Description

The va_start() function resets a variable argument list within a user-defined function statement so that the next va_arg() function call will return the first of the variable arguments. The va_start() function may be called multiple times within a function statement; each time it will reset the variable argument list to return the first argument.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following is an example of the va_start() function:

```
function reset_args(...) {
   term = va_start();
   i = 0;
   while (i++ <= 3) {
      printf("argument %d is %d\n",i,va_arg(term));
   }
   printf("variable arguments reset . . .\n");
   term = va_start();
   printf("reset argument is %d",va_arg(term));
   return;
}

function main() {
   reset_args(100,110,120,130,140,150,160);
   return;
}
```

The example produces the following output:

```
argument 1 is 100
argument 2 is 110
argument 3 is 120
variable arguments reset . . .
reset argument is 100
```

BMC Software, Inc., Confidential and Proprietary Information

write()

Write to a PSL process or file channel

Format

`write(`*`chan`*`,`*`text`*`)`

Parameters

Parameter	Definition
`chan`	process I/O channel number to which *text* is written
`text`	text to be written to channel *chan* The *text* can be a text string enclosed in double quotation marks, or one or more PSL commands that produce text as output.

Description

The write() function writes *text* to channel *chan*. The write() function returns the number of characters written or –1 on error.

If *text* cannot be written immediately, the write() function call blocks until it can either write the whole of *text* or the channel terminates.

Note

The write() function can block for a process channel created using the popen() function but not for a file channel created using the fopen() function.

BMC Software, Inc., Confidential and Proprietary Information

To enforce serialization for shared channels, no two reader processes (that is, the read() or readln() functions) can be blocked on the same channel. The second reader process that attempts to block on the shared channel will fail, returning the NULL string and setting the PSL variable errno to E_PSL_BUSY_CHANNEL.

Another possible shared channel failure can be caused by a close() function being executed against a channel that also has a blocked reader process.The close() function will cause the reader process to return the NULL string and set errno to E_PSL_UNBLOCKED_BY_CLOSE.

Example

The following PSL script uses the write() function to write to a disk file the table of decimal/octal/hexadecimal conversions that was created using the sprintf() function:

```
function main() {
   local channel,string;
   i = 0;
   channel = fopen("/local/convert.txt","w");
   if (channel != "") {
      string = sprintf(" n    Oct   Hex\n");
      string = sprintf("%s---  ----  ----\n",string);
      while (i++ <= 256) {
         string = sprintf("%s%3d  %4o  %4X\n",string,i,i,i);
      }
      write(channel,string);
      close(channel);
   } else {
      printf("File could not be opened, errno = %d",errno);
   }
   return;
}
```

BMC Software, Inc., Confidential and Proprietary Information

write_to_report()

Write arbitrary text to a popped-up task window

Format

`write_to_report(`*`handle`*`,`*`data`*`)`

Parameters

Parameter	Definition
`handle`	the task ID returned by a call to the popup_report() function
`data`	the text to be displayed in the task window

Description

The write_to_report() function writes arbitrary text to a task window that has been popped up on specified consoles. To pop up a task window, see the popup_report() function on page 1-269.

The write_to_report() function returns one of the following values:

- "1" if the write is successful
- "0" if the write fails

A write fails if all consoles have destroyed the task window thereby removing all the status information about the task. Thus, a subsequent call to popup_report() will pop up again the task window on the consoles.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following PSL script uses the write_to_report() function to write some text to a popped-up task window:

```
# Pop up a task window on all consoles that are monitoring the Foobar application
handle = popup_report("My Title","Foobar","","First bit of text\n");
ok = 1;
while (handle && ok) {
   sleep(10);
   # Pop up the task window on all consoles that have started monitoring the
   # Foobar application since the first call above (and previous call below)
   handle =  popup_report("My Second title","Foobar",handle,"Second bit of text\n");
   if (handle) { # the handle is still valid
      # Write some text to the task window
      write_to_report(handle,"hello world\n");
      ok = write_to_report(handle,"goodbye world\n");
  }
}
```

BMC Software, Inc., Confidential and Proprietary Information

BMC Software, Inc., Confidential and Proprietary Information

PSL Perform Agent Metric Functions

This chapter discusses the built-in PSL functions for the perform agent that are available on Windows. The following topics are discussed:

Introduction 2-2
Metrics and Metric Groups 2-2
Metric Data Types and Computations 2-3
closeMetric(). 2-7
closeMetricGroup() 2-10
disableMetric() 2-12
enableMetric() 2-14
enableMetricGroup() 2-16
findRowNumber() 2-18
getMetricComputation() 2-20
getMetricValue(). 2-23
getRowCount() 2-26
openMetric() 2-28
openMetricGroup(). 2-31
queryMetricComputations() 2-33
queryMetricGroups() 2-36
queryMetrics() 2-38
setMetricComputation() 2-40
updateMetricGroup() 2-43

BMC Software, Inc., Confidential and Proprietary Information

Introduction

The Perform Agent PSL functions are provided for PATROL users who use the PATROL perform agent. The Perform Agent PSL functions allow you to access data in the Perform Agent metric on a PATROL Agent for Windows.

Note

The Perform Agent PSL functions will not work with out the Perform Agent. For more information on the Perform Agent see your PATROL Perform documentation.

Metrics and Metric Groups

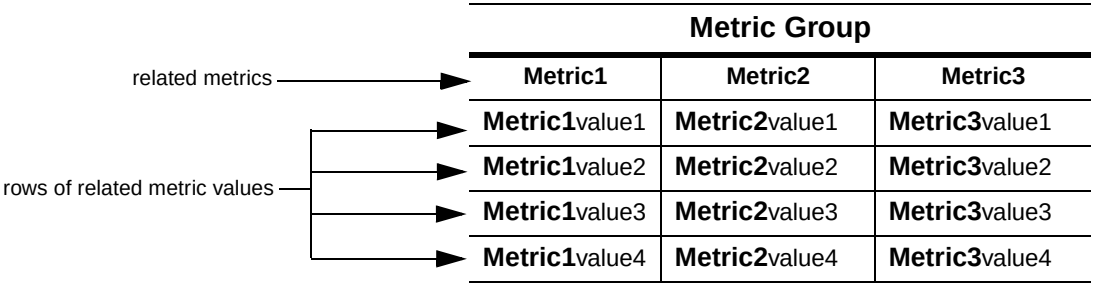
The Perform Agent Metric consists of metrics and metric groups. A metric is a collectible, measurable entity of data on a computer system or network. Examples of metrics are CPU Utilization, Total Disk I/O, and Disk Active Time.

A metric group is a group of related metrics. For example, System Statistics is a metric group that includes the metrics CPU Utilization, Memory Utilization, and Disk I/O.

Think of a metric group as a data table and a metric as a column in the table. The following figure shows the relation between metrics and metric groups:

BMC Software, Inc., Confidential and Proprietary Information

Figure 2-1 perform agent Metric Illustration



BMC supports well over a thousand metrics that can be collected, monitored, organized, and analyzed. See your PATROL Performance documentation for more information on the Perform metrics.

Metric Data Types and Computations

The perform agent collects metric data in several different raw data types, and depending on the data type, the data is computed to create various computations. The computations that are available for a metric depend on the raw or collected data. For example, character data does not create any additional computations–it is presented as raw data only.

Metric Data Types

The Perform metrics are classified into various data types. The data type refers to the data that was originally collected by the data collector. The metric data is collected in the following data types:

- character
- count
- float
- ID
- integer
- time
- utime

Table 2-1, “Metric Data Types,” on page 2-4 describes the raw data types supported by the perform agent.

BMC Software, Inc., Confidential and Proprietary Information

Table 2-1 Metric Data Types

Data Type	Description
character	text strings Character strings can be compared for alerting and filtering purposes.
count	a cumulative total since system boot time or process invocation
float	a float value that reflects data that is usually useful in its collected (raw) form
ID	identifier stored as number, such as process ID
integer	a 32–bit whole number, whose size is not dependent on machine word size
time	a cumulative count of time in seconds Similar to count data type, but used differently. Used to measure amount of time processor spends in certain state, disk spends being active, or process spends in system or user state.
utime	a Unix time format Distinct from time value format in that it is primarily used as a time stamp. (The utime value is resented in seconds since January 1, 1970.)

Note

The PSL functions do not display or manipulate the metric data types, but the data types determine the possible computations of the metric.

Metric Computations

The Perform Agent collects raw data, then depending on the data type, the agent creates various computations of the data. The following computations are available depending on the raw data type:

- % Time
- delta
- % Change
- Rate of Change
- Raw

Table 2-2 describes the metric computations and lists the raw data types supported by each computation type:

Table 2-2 Metric Computations and Supported Data Types

Computation	Description	Supported Data Types
% Time	determines percent of interval spent in given state **Example:** If metric is System CPU Time, % time refers to percent of data collection interval processor spent in system state.	time
Delta	reflects metric change during data collection interval	integer count float
% Change	reflects percent of change in raw metric value during data collection interval	integer count float
Rate of Change	determines how fast raw value of metric is changing during a period of time **Example:** If metric is Total Disk I/O, raw value is cumulative count of I/O since system boot. The rate of change is the average I/O rate during last data collection interval.	integer count float
Raw	denotes no computation is being performed on measured data	all data types

Metric Computation Equations

The Perform Agent metrics use standard equations to arrive at the metric computations. Table 2-3 lists the equations used for the metric computations:

Table 2-3 Metric Computation Equations (Part 1 of 2)

Computation	Equation
% Time	100 * (CV–PV)/(CT–PT)
Delta	CV–PV
% Change	100 * (CV–PV)/PV

BMC Software, Inc., Confidential and Proprietary Information

Table 2-3 Metric Computation Equations (Part 2 of 2)

Computation	Equation
Rate of Change	(CV–PV)/(CT–PT)
The following symbols are used for the metric computation equations: • CT–collection time of current data point • PV–previous value of the metric • CV–current value of the metric • PT–collection time of previous data point	

For more information on the Perform Agent and the Perform metric groups and metrics see your PATROL Performance documentation.

BMC Software, Inc., Confidential and Proprietary Information

closeMetric()

Close a handle to a metric

Format

`closeMetric(`*`mhandle`*`)`

Parameter

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.

Description

The closeMetric() function closes the handle to the metric, but the metric remains enabled for data collection. If you close a metric that is enabled, you will get the same metric handle when you call the openMetric() function to reopen the metric.

The closeMetric() function returns 1 for success and the null string (“”) on failure with the following errno value:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The closeMetric() function call is in bold.)

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric
closeMetric(mhandle);
```

BMC Software, Inc., Confidential and Proprietary Information

The example produces the following output:

```
The NT Process Statistics group has 39 rows of data,
and the Process Name metric has the following values:

row : value
===========
 01 : Idle
 02 : System
 03 : smss
 04 : csrss
 05 : winlogon
 06 : services
 07 : lsass
 08 : spoolss
 09 : RpcSs
 10 : msdtc
 11 : LexStart
 12 : AMGRSRVC
 13 : CLISVCL
 14 : inetd32
 15 : MCSHIELD
 16 : VSTSKMGR
 17 : METHWNT
 18 : Patrol
 19 : PatProcess
 20 : pstores
 21 : AtSvc
 22 : winmgmt
 23 : Wuser32
 24 : smsapm32
 25 : nddeagnt
 26 : Explorer
 27 : SysTray
 28 : PwsTray
 29 : loadwc
 30 : LAUNCH32
 31 : bgsagent
 32 : SHSTAT
 33 : tsystray
 34 : AcroTray
 35 : smsmon32
 36 : PatrolAgent
 37 : rundll32
 38 : bgscollect
 39 : bgs_sdservice.e
```

BMC Software, Inc., Confidential and Proprietary Information

closeMetricGroup()

Close the handle to the metric group and disable all metrics within the metric group

Format

`closeMetricGroup(`*`mghandle`*`)`

Parameter

Parameter	Definition
`mghandle`	the handle to a metric group The handle is returned by the openMetricGroup() function. See “openMetricGroup()” on page 2-31 for more information.

Description

The closeMetricGroup() function closes the handle to the metric group that was opened by the openMetricGroup() function. However, this function is different than the closeMetric() function because unlike the closeMetric() function, the closeMetricGroup() function disables all metrics within the group when the metric group handle is closed.

The closeMetric() function returns 1 for success and the null string (“”) on failure with the following errno value:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.

BMC Software, Inc., Confidential and Proprietary Information

Example

This example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The closeMetricGroup() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

disableMetric()

Discontinue data collection for a given metric

Format

`disableMetric(`*`mhandle`*`)`

Parameter

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.

Description

The disableMetric() function stops the data collection for the specified metric on the perform agent.

If the metric is not enabled, the disableMetric() function fails.

The disableMetric() function returns 1 for success and the null string (“”) on failure with one of the following errno values:

errno value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
130	E_PSL_DISABLED The specified metric is already disabled.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The disableMetric() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

enableMetric()

Start data collection for a metric within a specified metric group

Format

`enableMetric(`*`mhandle`*`)`

Parameter

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.

Description

The enableMetric() function starts the data collection for the specified metric on the perform agent. It is best to enable the metrics within a group before enabling the metric group.

If the metric is already enabled, the enableMetric() function fails.

The enableMetric() function returns 1 for success and the null string (“”) on failure with one of the following errno values:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
96	E_PSL_ENABLED The specified metric is already enabled.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The enableMetric() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

enableMetricGroup()

Start data collection for a specified metric group

Format

`enableMetricGroup(`*`mghandle`*`)`

Parameter

Parameter	Definition
`mghandle`	the handle to a metric group The handle is returned by the openMetricGroup() function. See “openMetricGroup()” on page 2-31 for more information.

Description

The enableMetricGroup() function starts the data collection for the specified metric group on the perform agent. It is best to enable the metrics within the metric group before enabling the metric group.

The metric group must be enabled before the metric group can be updated.

The enableMetricGroup() function returns 1 for success and the null string (“”) on failure with one of the following errno values:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
129	E_PSL_ENABLED The specified metric group is already enabled.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The enableMetricGroup() function call is in bold.) See the example output on page 2-9.

```
group="NT Process Statistics"; #define the metric group and metric
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

findRowNumber()

Find all rows within a metric that contain a given value

Format

`findRowNumber(`*`mhandle`*`,`*`value`*`)`

Parameters

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.
`value`	the value to match The value parameter is case sensitive for text values.

Description

The findRowNumber() function finds all rows in a metric that contain the specified *value*. The findRowNumber() function returns a list of row numbers in the specified metric that contain the specified *value*.

If the specified value is not found, the findRowNumber() function returns the null string (“”). On failure, the findRowNumber() function returns the null string (“”) with one of the following errno values:

errno value	Description
95	E_PSL_TIMEOUT There is no data available.
101	E_PSL_BAD_OBJECT An invalid handle was specified.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example checks the **Process Name** metric for the **Patrol** and **patrol** processes. (The findRowNumber() function call is in bold.)

```
group="NT Process Statistics"; #define the metric group and metric
metric="Process Name";
process_1="Patrol"; #define the process we want to find
process_2="patrol";
#open the metric group and metric
mghandle=openMetricGroup(group);
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#find the row numbers in the metric for the processes
row_1=findRowNumber(mhandle,process_1);
row_2=findRowNumber(mhandle,process_2);
#get the value of the returned row numbers
if (row_1==""){
   print (process_1." was not found in the ".metric." metric.\n\n");
} else {
   print(process_1." was found in row ".row_1.".\n\n");
   }
if (row_2==""){
   print (process_2." was not found in the ".metric." metric.\n\n");
} else {
   print("The ".process_2." process was found in row ".row_2.".\n\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

The example produces the following output:

```
Patrol was found in row 49.
patrol was not found in the Process Name metric.
```

BMC Software, Inc., Confidential and Proprietary Information

getMetricComputation()

Get the current metric computation for a metric

Format

`getMetricComputation(`*`mhandle`*`)`

Parameter

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.

Description

The getMetricComputation() function returns the computation that is currently being used for the metric specified by the *mhandle* parameter. Since each instance of a metricGroup(), created with the openMetricGroup() function, is unique, the same metric in different metric group instances can use different computations.

For more information on metric computations see “Metric Computations” on page 2-4.

Note

The getMetricComputation() function must be called before a metric is enabled with the enableMetric() function, or the getMetricComputation() function will fail.

The getMetricComputation() function returns the null string (“”) on failure with one of the following errno values:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
129	E_PSL_ENABLED The metric is already enabled. You must disable the metric to use this function.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the metrics in the **NT Logical Disk Configuration** metric group with their current computations. (The getMetricComputation() function call is in bold.)

```
#define the metric group and metric
group="NT Logical Disk Configuration";
#open the metric group and metric
mghandle=openMetricGroup(group);
#get a list of the metrics in the metric group
metricList=queryMetrics(mghandle);
print("The ".group." metric group:\n\n");
foreach metric (metricList){
   print(metric." => ");
   mhandle=openMetric(mghandle,metric);
   #get the computation for each metric
   comp=getMetricComputation(mhandle);
   print(comp."\n");
   closeMetric(mhandle);
}
#close the metric group
closeMetricGroup(mghandle);
```

The example produces the following output:

```
The NT Logical Disk Configuration metric group:

Drive Name => Raw
Drive Size => Raw
Paging File Name => Raw
Paging File init Size => Raw
Paging File Max Size => Raw
Paging File Current Size => Raw
Drive Free Space => Raw
Drive Utilization => Raw
```

BMC Software, Inc., Confidential and Proprietary Information

getMetricValue()

Get the data value for a metric on a given row

Format

`getMetricValue(`*`mhandle`*`,`*`row`*`)`

Parameters

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.
`row`	the row number in the metric to return the value of

Description

The getMetricValue() function returns the data value for a metric on a given row. The metric must be enabled with the enableMetric() function and updated with the updateMetricGroup() function before values can be retrieved using the getMetricValue() function.

While some metrics have several possible return values or computations, the getMetricValue() function will only return the current metric computation. See “setMetricComputation()” on page 2-40 for information on setting the metric computation.

BMC Software, Inc., Confidential and Proprietary Information

Additionally, keep in mind that even though the perform agent stores the metric values as different data types, like integers and strings, PSL is string based, and the getMetricValue() function will always return a string value.

Note

All 64-bit metric values are returned as two 32-bit hexadecimal values separated by a colon. The hexadecimal value can be converted to decimal using the PSL convert_base() function as in the following example:

```
hexnum64 = getMetricValue(mhandle,row);
decnum64 = convert_base("16","10",hexnum64);
```

See the “convert_base()” on page 1-65 for more information on the convert_base() function.

The getMetricValue() function returns the null string (“”) on failure with one of the following errno values:

errno Value	Description
95	E_PSL_TIMEOUT There is no data available.
101	E_PSL_BAD_OBJ An invalid handle was specified.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The getMetricValue() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

getRowCount()

Get the number of rows of data for the given data handle

Format

`getRowCount(mg`*`handle`*`)`

Parameter

Parameter	Definition
`mghandle`	the handle to a metric group The handle is returned by the openMetricGroup() function. See “openMetricGroup()” on page 2-31 for more information.

Description

The getRowCount() function returns the number of rows of data in the metric group specified by the *mghandle* parameter.

The data from a single collection cycle can contain a number of data rows. For example, if there are twenty processes on a system, the **NT Process Statistics** metric, will have twenty rows of data.

The getRowCount() function returns the null string (“”) on failure with one of the following errno values:

errno Value	Description
95	E_PSL_TIMEOUT There is no data available.
101	E_PSL_BAD_OBJ An invalid handle was specified.

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The getRowCount() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

openMetric()

Open a handle to a metric to access the metric

Format

`openMetric(`*`mghandle,metricName`*`)`

Parameters

Parameter	Definition
`mghandle`	the handle to a metric group The handle is returned by the openMetricGroup() function. See “openMetricGroup()” on page 2-31 for more information.
`metricName`	the name of the metric to open See “queryMetrics()” on page 2-38 for information on obtaining a list of available metrics.

Description

The openMetric() function opens a handle to the metric specified by the *metricName* parameter. The openMetric() function returns a handle to the metric that is used in other metric functions to refer to this instance of the metric.

Once a handle is opened to a metric, it can be enabled for data collection. You can open different instances of a metric using different metric group handles.

The openMetric() function returns the null string (“”) on failure with one of the following errno values:

BMC Software, Inc., Confidential and Proprietary Information

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
132	E_PSL_BAD_METRC The specified metric does not exist.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The openMetric() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

openMetricGroup()

Open a handle to a metric group to access the metric group

Format

`openMetricGroup(`*`mgroupName`*`)`

Parameters

Parameter	Definition
mgroupName	the name of the metric group to open See “queryMetricGroups()” on page 2-36 for information on obtaining a list of available metrics.

Description

The openMetricGroup() function returns a handle to the metric group. You can open a metric group more than once, and each time you open a metric group, you receive a new handle, so you can have multiple instances of a metric group.

Once you open a handle to a metric group, you can enable it for data collection.

The openMetricGroup() function returns the null string (“”) on failure with one of the following errno values:

errno Value	Description
121	E_PSL_ALLOC_FAILED The function was unable to create a handle to the group.
131	E_PSL_BAD_METRIC_GROUP The metric group specified is not valid.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The openMetricGroup() function call is in bold.) See the example output on page 2-9.

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

queryMetricComputations()

Get a list of available computations for a metric

Format

`queryMetricComputations(`*`mhandle`*`)`

Parameters

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.

Description

The queryMetricComputations() function returns a list of available computations for the metric. See “Metric Data Types and Computations” on page 2-3 for more information on metric computations.

Note

The queryMetricComputations() function must be called before a metric is enabled with the enableMetric() function, or the queryMetricComputations() function will fail.

BMC Software, Inc., Confidential and Proprietary Information

The queryMetricComputations() function returns the null string (“”) on failure with one of the following errno values:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
129	E_PSL_ENABLED The metric is already enabled. You must disable the metric to use this function.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints a list of the metrics and their available computations for the **NT Disk Mapping Configuration** metric group. (The queryMetricComputations() function call is in bold.)

```
#define the metric group and metric
group="NT Disk Mapping Configuration";
#open the metric group and metric
mghandle=openMetricGroup(group);
#get a list of the metrics in the metric group
metricList=queryMetrics(mghandle);
print("The ".group." metric group:\n\n");

foreach metric (metricList){
   print(metric."\n=====================================\n");
   mhandle=openMetric(mghandle,metric);
   #get the list of computations for each metric
   compList=queryMetricComputations(mhandle);
   print(compList."\n\n");
   closeMetric(mhandle);
}
#close the metric group
closeMetricGroup(mghandle);
```

This example produces the following output:

```
The NT Disk Mapping Configuration metric group:

Logical Drive Name
=====================================
Raw

Physical Disk Name
=====================================
Raw

Physical Partition size
=====================================
Delta
% Change
Rate of Change
Raw
```

BMC Software, Inc., Confidential and Proprietary Information

queryMetricGroups()

Return a list of available metric groups

Format

```
queryMetricGroups([group])
```

Parameter

Parameter	Definition
`group`	optional metric group name If the metric group name is specified, the queryMetricGroups() function returns the name of the group if it exists. If the group name is not specified, the queryMetricGroups() function returns all metric groups defined on the perform agent.

Description

The queryMetricGroups() function returns a list of defined metric groups, or the name of the specified metric group. If a metric group name is returned by the queryMetricGroup() function, it means that it is a valid metric group name. However, the metric group may not be available on the perform agent, and it may not be enabled.

The queryMetricGroups() function returns the null string (“”) if no metric groups are found and the null string on failure with the following errno value:

errno Value	Description
121	E_PSL_ALLOC_FAILED The function could not obtain a consumer instance.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints a list of all the metrics known by the perform agent. (The queryMetricGroups() function call is in bold.)

```
#get a list of the metric groups
metricGroupList=queryMetricGroups();

#print the list of metrics in the group
print(metricGroupList."\n\n");
```

The example produces the following output:

```
NT System Statistics
NT Processor Statistics
NT Memory Statistics
NT Cache Statistics
NT Physical Disk Statistics
NT Logical Disk Statistics
NT Process Statistics
NT Thread Statistics
NT Objects Statistics
NT Redirector Statistics
NT Disk Controller Configuration
NT User Configuration
NT Network Adapter Configuration
NT IP Configuration
NT MS Exchange Client Configuration
NT Extensible PerfCounter Configuration
UMX Command
UMX Node Status
.
.
.
NT Port Statistics
NT NETBIOS Statistics
NT Service Statistics
NT Configuration Statistics
UMX Directory
UMX File Statistics
UMX Active Policies
UMX Collect Requests
UMX Log File Reader
NT Application Log Statistics
NT System Log Statistics
NT Security Log Statistics
```

BMC Software, Inc., Confidential and Proprietary Information

queryMetrics()

Return a list of available metrics in the given metric group

Format

`queryMetrics(`*`mghandle`*`,[`*`metric`*`])`

Parameter

Parameter	Definition
`mghandle`	the handle to a metric group
`metric`	optional metric name If the *metric* name is specified, the queryMetrics() function returns the name of the metric if it exists in the specified metric group. If the *metric* name is not specified, the queryMetrics() function returns all metrics defined for the metric group.

Description

The queryMetrics() function returns a list of available metrics within the specified metric group or the name of the metric specified. The metrics in the list must be enabled using the enableMetric() function before data can be collected from the metrics using the getMetricValue() function.

The queryMetrics() function returns the null string (“”) if the metric can not be found and the null string on failure with the following errno value:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.

Example

The following example prints the metrics in the **NT Disk Mapping Configuration**. (The queryMetrics() function call is in bold.)

```
#define the metric group and metric
group="NT Disk Mapping Configuration";

#open the metric group and metric
mghandle=openMetricGroup(group);

#get a list of the metrics in the metric group
metricList=queryMetrics(mghandle);

print("The ".group." metric group \nhas the following metrics:\n\n");

#print the list of metrics in the group
print(metricList."\n\n");

#close the metric group
closeMetricGroup(mghandle);
```

The example produces the following output:

```
The NT Disk Mapping Configuration metric group
has the following metrics:

Logical Drive Name
Physical Disk Name
Physical Partition size
```

BMC Software, Inc., Confidential and Proprietary Information

setMetricComputation()

Set the metric computation for a metric

Format

`setMetricComputation(`*`mhandle`*`,`*`computationName`*`)`

Parameters

Parameter	Definition
`mhandle`	the handle to a metric The handle is returned by the openMetric() function. See “openMetric()” on page 2-28 for more information.
`computationName`	the name of the computation to use for the metric **Valid Values** % Time delta % Change Rate of Change Raw The valid computation is dependent on the metric and data type. For more information on computation types, see “Metric Data Types and Computations” on page 2-3.

BMC Software, Inc., Confidential and Proprietary Information

Description

The setMetricComputation() function sets the computation for the metric specified by the *mhandle* parameter.

When the metric computation is set using the setMetricComputation() function, the computation only applies to the metric specified by the metric handle. If you open a metric group twice you receive two different metric group handles, and if you open the same metric using the two different metric group handles, you have two different instances of the metric and can set different computations for each metric using the setMetricComputation() function.

Note

The setMetricComputation() function must be called before a metric is enabled with the enableMetric() function, or the setMetricComputation() function will fail. You must set the computation you want to use before enabling the metric.

The setMetricComputation() function returns 1 for success and the null string (“”) on failure with one of the following errno values:

errno Value	Description
101	E_PSL_BAD_OBJ An invalid handle was specified.
129	E_PSL_ENABLED The metric is already enabled. You must disable the metric to use this function.
133	E_PSL_BAD_COMPUTATION An invalid computation was specified for the metric.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example get a list of possible metric computations for the **NT Disk Mapping Configuration** metric, and then sets the computation to each value. (The setMetricComputation() function call is in bold.)

```
#define the metric group and metric
group="NT Disk Mapping Configuration";
metric="Physical Partition size";
#open the metric group and metric
mghandle=openMetricGroup(group);
mhandle=openMetric(mghandle,metric);
#get a list of computations for the metric
compList=queryMetricComputations(mhandle);
print("The ".metric." metric has the following computations:\n");
print("======================\n".compList."\n======================\n\n");
print("Setting the metric computation to each value:\n\n");
foreach comp(compList){
   #set the current computation to each value in the list
   setMetricComputation(mhandle,comp);
   currentComp=getMetricComputation(mhandle);
   print("We are using the ".currentComp." computation.\n\n");
}
#close the metric and metric group
closeMetric(mhandle);
```

The example produces the following output:

```
The Physical Partition size metric has the following computations:
======================
Delta
% Change
Rate of Change
Raw
======================

Setting the metric computation to each value:

We are using the Delta computation.

We are using the % Change computation.

We are using the Rate of Change computation.

We are using the Raw computation.
```

BMC Software, Inc., Confidential and Proprietary Information

updateMetricGroup()

Update the data for a metric group

Format

`updateMetricGroup(`*`handle`*`)`

Parameter

Parameter	Definition
`handle`	the handle of the metric group or metric that is updated The handle is returned by the openMetricGroup() or openMetric() function.

Description

The updateMetricGroup() function refreshes the data for the metric or metric group specified by the *handle* parameter.

You must ensure that the metric group enables successfully using the enableMetricGroup() function before attempting to update the metric group for data collection, or the updateMetricGroup() function will fail.

Note

A delay in returning data can cause updateMetricGroup() to return errno 95 E_PSL_TIMEOUT. Call this function within a loop to give it some extra time to get good data.

BMC Software, Inc., Confidential and Proprietary Information

The updateMetricGroup() function returns 1 for success and the null string (“”) on failure with one of the following errno values:

errno Value	Description
95	E_PSL_TIMEOUT There is no data available.
101	E_PSL_BAD_OBJ An invalid handle was specified.
130	E_PSL_DISABLED The metric group is disabled.

BMC Software, Inc., Confidential and Proprietary Information

Example

The following example prints the **Process Name** metric value for each row of the **NT Process Statistics** metric group. (The updateMetricGroup() function call is in bold.)

```
#define the metric group and metric
group="NT Process Statistics";
metric="Process Name";
#open the metric group
mghandle=openMetricGroup(group);
#open a metric in the metric group
mhandle=openMetric(mghandle,metric);
#enable the metric and metric group for data collection
enableMetric(mhandle);
enableMetricGroup(mghandle);
#Call updateMetricGroup(} in loop to prevent errno 95 E_PSL_TIMEOUT
for(x=0;((x<10) && (!done));x++){
   result = updateMetricGroup(mghandle);
   if (result){done = 1;} #update metric worked get out of the loop
   else{sleep(1);} # update metric failed sleep and call the method again.
}#End for loop
#get the number of rows in the metric group
count=getRowCount(mghandle);
#print the description of the metric data output
print("The ".group." group has ".count. " rows of data,\n");
print("and the ".metric." metric has the following values:");
print("\n\nrow : value\n===========\n");

#get the metric value for each row and print it with the row number
for(row=1; row<=count; row++){
   #make the row an integer to remove the decimal point for display
   row=int(row);
   #if the row number is less than 10 add a zero to the front for formatting
   if (row < 10) {row ="0".row;}
   #get the value for the metric for the current row
   value=getMetricValue(mhandle, row);
   #print the row number and metric value
   print(" ".row." : ".value."\n");
   }
#discontinue data collection for the metric
disableMetric(mhandle);
#close the metric and metric group
closeMetric(mhandle);
closeMetricGroup(mghandle);
```

BMC Software, Inc., Confidential and Proprietary Information

The example produces the following output:

```
The NT Process Statistics group has 39 rows of data,
and the Process Name metric has the following values:

row : value
===========
 01 : Idle
 02 : System
 03 : smss
 04 : csrss
 05 : winlogon
 06 : services
 07 : lsass
 08 : spoolss
 09 : RpcSs
 10 : msdtc
 11 : LexStart
 12 : AMGRSRVC
 13 : CLISVCL
 14 : inetd32
 15 : MCSHIELD
 16 : VSTSKMGR
 17 : METHWNT
 18 : Patrol
 19 : PatProcess
 20 : pstores
 21 : AtSvc
 22 : winmgmt
 23 : Wuser32
 24 : smsapm32
 25 : nddeagnt
 26 : Explorer
 27 : SysTray
 28 : PwsTray
 29 : loadwc
 30 : LAUNCH32
 31 : bgsagent
 32 : SHSTAT
 33 : tsystray
 34 : AcroTray
 35 : smsmon32
 36 : PatrolAgent
 37 : rundll32
 38 : bgscollect
 39 : bgs_sdservice.e
```

BMC Software, Inc., Confidential and Proprietary Information

Index

A

acos() arccosine 9
ActiveX scripts 154
add_diary() add text to event diary 11
annotate() annotate parameter value 13
annotate_get() return parameter annotation 19
application discovery functions
    full_discovery() verify mode 176
    in_transition() allow state change 206
    proc_exists() verify process 280
    process() process status 281
asctime() date/time 22
asin() arcsine 26
atan() arctangent 27
atexit() call at end of process 28

B

batch_set() set variables 30
blackout() hide state changes 33
built-in functions 138
    acos() arccosine 9
    add_diary() add text to event diary 11
    annotate() annotate parameter value 13
annotate_get() return parameter annotation 19
asctime() date/time 22
asin() arcsine 26
atan() arctangent 27
atexit() call at end of process 28
batch_set() set variables 30
blackout() hide state changes 33
cat() return file contents 38
ceil() integer ceiling 40
chan_exists() verify channel 41
change_state() change object state 42
chart() load/destroy/print chart 45
close() close channel 54
closeMetric() 7
closeMetricGroup() 10
code_cvt() convert a string from one codeset to another 56
cond_signal() signal process 59
cond_wait() process wait 60
console_type() identify PATROL Developer Console 64
convert_base() convert base 65
convert_date() return system time 68
convert_locale_date() internationalized version of convert_date() 74
cos() cosine 78
cosh() hyperbolic cosine 81

BMC Software, Inc., Confidential and Proprietary Information

create() create object 83
date() date/time 88
dcget_text() get message using domain and category 89
debugger() suspend for debug 91
desktop() execute PATROL link command 92
destroy() destroy object 93
destroy_lock() destroy process lock 94
dget_text() get message using domain 95
difference() unique to list 97
disableMetric() 12
dump_hist() history data 99
enableMetric() 14
enableMetricGroup() 16
encrypt() encrypt a string 103
event_archive() archive PATROL events 105
event_catalog_get() return catalog information 111
event_check() count matching events 114
event_query() print matching events 119
event_range_manage() change status of event range 127
event_range_query() return events from an ID range 129
event_report() return event report 132
event_trigger() create PATROL event 146
event_trigger2() create PATROL event 149
execute() submit command 152
exists() verify object 156
exp() e to a power 158
fabs() absolute value 159
file() file information 160
findRowNumber() 18
floor() integer floor 164
fmod() floating point remainder 165
fopen() open file channel 166
fseek() set file position 171
ftell() file position 174
full_discovery() verify mode 176
get() get value 177
get_chan_info() channel status 178
get_ranges() return alarm range settings 182
get_text() get message text 186
get_vars() list variables 188
getenv() object environment 190
getMetricComputation() 20
getMetricValue() 23
getpid() return process ID 192
getpname() get process name 193
getRowCount() 26
grep() match expression 194
history() return history 198
history_get_retention() return retention value 201
in_transition() allow state change 206
index() locate string 202
int() return integer 203
internal() process internal to PATROL Agent 204
intersection() set intersection 205
is_var() verify variable 210
isnumber() verify expression 208
join() join strings together 212
kill() terminate PSL process 213
length() count characters 214
lines() count lines 215
lock() request lock 216
log() record event 219
log10() logarithm (10) 222
loge() logarithm (e) 220
matchline() Save the results of a regular-expression to variables 224
msg_check() validate a message 227

BMC Software, Inc., Confidential and Proprietary Information

msg_get_format() return format information for a message 229
msg_get_severity() get severity of a message 231
msg_printf() display message 233
msg_sprintf() return message 235
ntharg() return text match 237
nthargf() return text match 240
nthline() return lines 243
nthlinef() return lines 246
num_bytes() return string length in bytes 248
num_consoles() number of registered consoles 251
openMetric() 28
openMetricGroup() 31
pconfig() configure PATROL Agent 253
popen() process channel 265
poplines() return a number of lines 267
pow() raise to a power 272
print() formatted output 274
printf() C output format 275
proc_exists() verify process 280
process() process status 281
PslExecute() start PSL process 284
PslSetOptions() set run-time options 287
queryMetricComputations() 33
queryMetricGroups() 36
queryMetrics() 38
random() random numbers 288
read() channel read 290
readln() read line 294
refresh_parameters() execute PATROL parameters 296
remote_close() end remote agent session 298
remote_event_query() return remote agent events 299
remote_event_trigger() create remote agent event 306
remote_file_send() copy file to remote agent 309
remote_open() open remote agent session 315
remove() remove system file 323
replace() replace string in text 325
response() create dialog 326
response_get_value() return dynamic dialog status 401
rindex() last occurrence 404
sec_store_get() get value from secure area of a PATROL Agent 405
sec_store_set() store and encrypt a value in the PATROL Agent 407
set() set variable 410
set_alarm_ranges() set alarm conditions for a parameter 412
set_locale() set or get the value of a locale category 416
setMetricComputation() 40
share() make global channel 419
sin() sine function 421
sinh() hyperbolic sine 424
sleep() suspend process 427
snmp_agent_config() configure SNMP agent 429
snmp_agent_start() start SNMP agent 431
snmp_agent_stop() stop SNMP agent 433
snmp_close() close SNMP session 434
snmp_config configure SNMP session 435
snmp_get() return MIB variables 441
snmp_get_next() get next MIB variable 444
snmp_h_get() get MIB variable fast 447
snmp_h_get_next() return next variable fast 450
snmp_h_set() set MIB variables fast 453

BMC Software, Inc., Confidential and Proprietary Information

snmp_open() open SNMP session 456
snmp_set() set MIB variables 459
snmp_trap_ignore() ignore SNMP traps 462
snmp_trap_listen() accumulate SNMP traps 463
snmp_trap_raise_std_trap() raise PATROL trap 466
snmp_trap_receive() receive SNMP traps 467
snmp_trap_register_im() PATROL trap receiver list 472
snmp_trap_send() send SNMP trap 476
snmp_walk() subsequent MIB variables 479
sort() sort list 481
splitline() parse text into variables 484
sprintf() print as string 487
sqrt() square root 493
srandom() seed random number 495
subset() verify subset 496
substr() return substring 498
system() submit command 499
tail() last text lines 501
tan() tangent 503
tanh() hyperbolic tangent 505
text_domain() set or get current domain 507
time() seconds since 01/01/70 509
tmpnam() temporary file name 510
tolower() all lower case 511
toupper() all upper case 512
trace_psl_process() enable tracing 513
trim() remove characters 515
union() merge lists 518
unique() remove duplicate elements 520
unlock() release lock 521
unset() remove set() object 523
updateMetricGroup() 43
va_arg() return next argument 524
va_start() reset argument list 526
write() write to channel 528

C

cat() return file contents 38
ceil() integer ceiling 40
chan_exists() verify channel 41
change_state() change object state 42
chart() load, destroy, print a chart 45
check boxes 361
clicker 371
See also time spinner
close() close channel 54
closeMetric() 7
closeMetricGroup() 10
code_cvt() convert a string from one codeset to another 56
columns, in dialog boxes 384
combined state 43
command button, *See* popup dialog boxes
command functions
chan_exists() verify channel 41
close() close channel 54
desktop() execute PATROL link command 92
execute() submit command 152
fopen() open file channel 166
get_chan_info() channel status 178
internal() process internal to PATROL Agent 204
popen() process channel 265
share() make a global channel 419
system() submit command 499
compound control elements, in dialog boxes 353
computations 4
cond_signal() signal process 59
cond_wait() process wait 60
condition functions

BMC Software, Inc., Confidential and Proprietary Information

cond_signal signal process 59
cond_wait() process wait 60
console functions
response_get_value() return dynamic dialog status 401
console_type() identify PATROL Developer Console 64
convert_base() convert base 65
convert_date() return system time 68
convert_locale_date() internationalized version of convert_date() 74
cos() cosine 78
cosh() hyperbolic cosine 81
create() create object 83

D

data 2
data types 3
metrics 2
data types 3
date() date/time 88
dcget_text() get message using domain and category 89
debugger() suspend for debug 91
debugging functions
debugger() suspend for debug 91
designing dialog boxes 331
desktop() execute PATROL link command 92
destroy() destroy object 93
destroy_lock() destroy process lock 94
dget_text() get message using domain 95
dialog boxes
controls for 351
designing 331
nested 387
popups in 387
difference() unique to list 97
disableMetric() 12
dump_hist() history data 99

E

enableMetric() 14
enableMetricGroup() 16
encrypt() encrypt a string 103
equations 5
event functions
add_diary() add text to event diary 11
event_archive() archive PATROL events 105
event_catalog_get() return catalog information 111
event_check() count matching events 114
event_query() print matching events 119
event_range_manage() change status of event range 127
event_range_query() return events from an ID range 129
event_report() return event report 132
event_schedule() schedule PATROL events 138
event_trigger() create PATROL event 146
event_trigger2() create PATROL event 149
event_archive() archive PATROL events 105
event_catalog_get() return catalog information 111
event_check() count matching events 114
event_query() print matching events 119
event_range_manage() change status of event range 127
event_range_query() return events from an ID range 129
event_report() return event report 132
event_schedule() schedule PATROL events 138

BMC Software, Inc., Confidential and Proprietary Information

event_trigger() create PATROL event 146
event_trigger2() create PATROL event 149
execute() submit command 152
exists() verify object 156
exp() e to a power 158

F

fabs() absolute value 159
file functions
    cat() return file contents 38
    file() file information 160
    fseek() set file position 171
    ftell() file position 174
    remove() remove system file 323
file() file information 160
findRowNumber() 18
floor() integer floor 164
fmod() floating point remainder 165
fopen() open file channel 166
frame 382
fseek() set file position 171
ftell() file position 174
full_discovery() verify mode 176

G

get() get value 177
get_chan_info() channel status 178
get_ranges() return alarm range settings 182
get_text() get message text 186
get_vars() list variables 188
getenv() object environment 190
getMetricComputation() 20
getMetricValue() 23
getpid() return process ID 192
getpname() get process name with pid 193
getRowCount() 26
grep() match expression 194
group box, *See* frame

H

history() return history 198
history_get_retention() return retention value 201

I

I/O functions
    chart() load/destroy/print chart 45
    log() record event 219
    print() formatted output 274
    printf() C output format 275
    read() channel read 290
    readln() read line 294
    sprintf() print as string 487
    tmpnam() temporary file name 510
    write() write to channel 528
icons in dialog boxes 378
in_transition() allow state change 206
index() locate string 202
int() return integer 203
internal() process internal to PATROL Agent 204
international (ID-based messaging) functions
    msg_check() validate a message 227
    msg_get_format() return format information for a message 229
    msg_get_severity() get severity of a message 231
    msg_printf() display message 233
    msg_sprintf() return message 235
international (string-based messaging) functions
    dcget_text() get message using domain and category 89

BMC Software, Inc., Confidential and Proprietary Information

dget_text() get message using domain 95
get_text() get message text 186
text_domain() set or get current domain 507
international functions
code_cvt() convert a string from one codeset to another 56
convert_locale_date() internationalized version of convert_date() 74
num_bytes() return string length in bytes 248
set_locale() set or get the value of a locale category 416
intersection() set intersection 205
is_var() verify variable 210
isnumber() verify expression 208

J

join() join strings together 212

K

kill() terminate PSL process 213

L

label 377
layout of dialog boxes 331
length() count characters 214
lines() count lines 215
list
box 366
multiple-selection 366
single-selection 366
lock functions
destroy_lock() destroy process lock 94
lock() request lock 216
lock() request lock 216
locking functions
unlock() release lock 521
log() record event 219
log10() logarithm (10) 222
loge() logarithm (e) 220

M

matchline() Save the results of a regular-expression to variables 224
mathematical functions
acos() arccosine 9
asin() arcsine 26
atan() arctangent 27
ceil() integer ceiling 40
convert_base() convert base 65
cos() cosine 78
cosh() hyperbolic cosine 81
exp() e to a power 158
fabs() absolute value 159
floor() integer floor 164
fmod() floating point remainder 165
log10() logarithm 10 222
loge() logarithm (e) 220
pow() raise to a power 272
random() random numbers 288
sin() sine function 421
sinh() hyperbolic sine 424
sqrt() square root 493
srandom() seed random number 495
tan() tangent 503
tanh() hyperbolic tangent 505
metric groups 2
metrics 2
msg_check() validate a message 227
msg_get_format() return format information for a message 229

BMC Software, Inc., Confidential and Proprietary Information

msg_get_severity() get severity of a message 231
msg_printf() display message 233
msg_sprintf() return message 235
multiple-selection list 366

N

nested dialog boxes
    described 387
ntharg() return text match 237
nthargf() return text match 240
nthline() return lines 243
nthlinef() return lines 246
num_bytes() return string length in bytes 248
num_consoles() number of registered consoles 251
numeric functions 203

O

object function
    exists() verify object 156
object functions 83
    batch_set() set variables 30
    blackout() hide state changes 33
    change_state() change object state 42
    destroy() destroy object 93
    get() get value 177
    get_vars() list variables 188
    getenv() object environment 190
    is_var() verify variable 210
    set() set variable 410
    sort() sort list 481
    unset() remove set() object 523
object state 43
openMetric() 28
openMetricGroup() 31
option buttons, *See* radio buttons
option menu 369

P

parameter functions
    annotate() annotate parameter value 13
    annotate_get() return parameter annotation 19
    get_ranges() return alarm range settings 182
    refresh_parameters() execute PATROL parameters 296
    set_alarm_ranges() set alarm conditions for a parameter 412
parameter history functions
    dump_hist() history data 99
    history() return history 198
    history_get_retention() return retention value 201
PATROL Agent functions
    pconfig() configure PATROL Agent 253
PATROL Console functions
    console_type() identify PATROL Developer Console 64
    num_consoles() number of registered consoles 251
    response() create dialog 326
pconfig() configure PATROL Agent 253
Perform Agent 2
    computations 4
    data types 3
    equations 5
    metric groups 2
    metrics 2
Perform Agent functions
    closeMetric() 7
    closeMetricGroup() 10
    disableMetric() 12
    enableMetric() 14

BMC Software, Inc., Confidential and Proprietary Information

enableMetricGroup() 16
findRowNumber() 18
getMetricComputation() 20
getMetricValue() 23
getRowCount() 26
openMetric() 28
openMetricGroup() 31
queryMetricComputations() 33
queryMetricGroups() 36
queryMetrics() 38
setMetricComputation() 40
updateMetricGroup() 43
Performance
Perform Agent 2
popen() process channel 265
poplines() return a number of lines 267
popup dialog boxes
described 387
popup_report() popup task window 269
pow() raise to a power 272
print() formatted output 274
printf() C output format 275
proc_exists() verify process 280
process control functions
atexit() call at end of process 28
getpid() return process ID 192
getpname() get process name 193
kill() terminate PSL process 213
PslExecute() start PSL process 284
PslSetOptions() set run-time options 287
sleep() suspend process 427
trace_psl_process() enable tracing 513
process() process status 281
PslExecute() start PSL process 284
PslSetOptions() set run-time options 287

Q

queryMetricComputations() 33
queryMetricGroups() 36
queryMetrics() 38

R

radio boxes, *See* radio buttons
radio buttons 359
random() random numbers 288
read() channel read 290
readln() read line 294
refresh_parameters() execute PATROL parameters 296
remote functions
remote_close() end remote agent session 298
remote_event_query() return remote agent events 299
remote_event_trigger() create remote agent event 306
remote_file_send() copy file to remote agent 309
remote_open() open remote agent session 315
remote_close() end remote agent session 298
remote_event_query() return remote agent events 299
remote_event_trigger() create remote agent event 306
remote_file_send() copy file to remote agent 309
remote_open() open remote agent session 315
remove() remove system file 323
replace() replace string in text 325
response() create dialog 326
compound element nesting 352, 353, 354
control field summary 351
field formats
response() high-level 326

BMC Software, Inc., Confidential and Proprietary Information

return value formats 331
response_get_value() return dynamic dialog status 401
rindex() last occurrence 404
rows, in dialog boxes 386
rule state 43

S

sec_store_get() get value from secure area of a PATROL Agent 405
sec_store_set() store and encrypt a value in the PATROL Agent 407
secondary dialog boxes
described 387
security functions
sec_store_get() get value from secure area of a PATROL Agent 405
sec_store_set() store and encrypt a value in the PATROL Agent 407
separator 380
set functions
difference() unique to set 97
intersection() set intersection 205
subset() verify subset 496
union() merge lists 518
unique() remove duplicate elements 520
set() set variable 410
set_alarm_ranges() set alarm conditions for a parameter 412
set_locale() set or get the value of a locale category 416
setMetricComputation() 40
share() make global channel 419
share() share channel
KM developer notes 420
simple controls, in dialog boxes 352
sin() sine function 421
single-selection list 366
sinh() hyperbolic sine 424
sleep() suspend process 427
slider, *See* sliding scale
sliding scale 375
SNMP functions
snmp_agent_config() configure SNMP agent 429
snmp_agent_start() start SNMP agent 431
snmp_agent_stop() stop SNMP agent 433
snmp_close() close SNMP session 434
snmp_config() configure SNMP session 435
snmp_get() return MIB variables 441
snmp_get_next() get next MIB variable 444
snmp_h_get() get MIB variable fast 447
snmp_h_get_next() return next variable fast 450
snmp_h_set() set MIB variables fast 453
snmp_open() open SNMP session 456
snmp_send_trap() send SNMP trap 476
snmp_set() set MIB variables 459
snmp_trap_ignore() ignore SNMP traps 462
snmp_trap_listen() accumulate SNMP traps 463
snmp_trap_raise_std_trap() raise PATROL trap 466
snmp_trap_receive() receive SNMP traps 467
snmp_trap_register_im() PATROL trap receiver list 472
snmp_walk() subsequent MIB variables 479
snmp_agent_config() configure SNMP agent 429
snmp_agent_start() start SNMP agent 431
snmp_agent_stop() stop SNMP agent 433
snmp_close() close SNMP session 434

BMC Software, Inc., Confidential and Proprietary Information

snmp_config() configure SNMP session 435
_snmp_debug() start SNMP debugging 440
built-in functions
    _snmp_debug() start SNMP debugging 440
SNMP functions
    _snmp_debug() start SNMP debugging 440
snmp_get() return MIB variables 441
snmp_get_next() get next MIB variable 444
snmp_h_get() get MIB variable fast 447
snmp_h_get_next() return next variable fast 450
snmp_h_set() set MIB variables fast 453
snmp_open() open SNMP session 456
snmp_set() set MIB variables 459
snmp_trap_ignore() ignore SNMP traps 462
snmp_trap_listen() accumulate SNMP traps 463
snmp_trap_raise_std_trap() raise PATROL trap 466
snmp_trap_receive() receive SNMP traps 467
snmp_trap_register_im() PATROL trap receiver list 472
snmp_trap_send() send SNMP trap 476
snmp_walk() subsequent MIB variables 479
sort() sort list 481
spin box, *See* clicker
split line, *See* separator
splitline() parse text into variables 484
sprintf() print as string 487
sqrt() square root 493
srandom() seed random number 495
state 43
string functions
    encrypt() encrypt a string 103
    grep() match expression 194
    index() locate string 202
    isnumber() verify expression 208
    join() join strings together 212
    length() count characters 214
    lines() count lines 215
    matchline() Save the results of a regular-expression to variables 224
    ntharg() return text match 237
    nthargf() return text match 240
    nthline() return lines 243
    nthlinef() return lines 246
    poplines() return a number of lines 267
    replace() replace string in text 325
    rindex() last occurrence 404
    splitline() parse text into variables 484
    substr() return substring 498
    tail() last text lines 501
    tolower() all lower case 511
    toupper() all upper case 512
    trim() remove characters 515
subset() verify subset 496
substr() return substring 498
system() submit command 499

T

tail() last text lines 501
tan() tangent 503
tanh() hyperbolic tangent 505
text_domain() set or get current domain 507
time functions
    asctime() date/time 22
    convert_date() return system time 68
    date() date/time 88
    time() seconds since 01/01/70 509
time spinner 373
    See also clicker
time() seconds since 01/01/70 509
tmpnam() temporary file name 510
toggle button 364
tolower() all lower case 511
toupper() all upper case 512

BMC Software, Inc., Confidential and Proprietary Information

trace_psl_process() enable tracing 513
trackbar control, *See* sliding scale
trim() remove characters 515

U

unfold button, *See* popup dialog boxes
union() merge lists 518
unique() remove duplicate elements 520
unlock() release lock 521
unset() remove set() object 523
updateMetricGroup() 43

V

va_arg() return next argument 524
va_start() reset argument list 526
variable argument functions
    va_arg() return next argument 524
    va_start() reset argument list 526

W

write() write to channel 528

BMC Software, Inc., Confidential and Proprietary Information

STOP!

IMPORTANT INFORMATION - DO NOT INSTALL THIS PRODUCT UNLESS YOU HAVE READ ALL OF THE FOLLOWING MATERIAL

By clicking the YES or ACCEPT button below (when applicable), or by installing and using this Product or by having it installed and used on your behalf, You are taking affirmative action to signify that You are entering into a legal agreement and are agreeing to be bound by its terms, EVEN WITHOUT YOUR SIGNATURE. BMC is willing to license this Product to You ONLY if You are willing to accept all of these terms. CAREFULLY READ THIS AGREEMENT. If You DO NOT AGREE with its terms, DO NOT install or use this Product; press the NO or REJECT button below (when applicable) or promptly contact BMC or your BMC reseller and your money will be refunded if by such time You have already purchased a full-use License.

SOFTWARE LICENSE AGREEMENT FOR BMC PRODUCTS

SCOPE. This is a legally binding Software License Agreement (**"License"**) between You (either an individual or an entity) and BMC pertaining to the original computer files (including all computer programs and data stored in such files) contained in the enclosed Media (as defined below) or made accessible to You for electronic delivery, if as a prerequisite to such accessibility You are required to indicate your acceptance of the terms of this License, and all whole or partial copies thereof, including modified copies and portions merged into other programs (collectively, the **"Software"**). **"Documentation"** means the related hard-copy or electronically reproducible technical documents furnished in association with the Software, **"Media"** means the original BMC-supplied physical materials (if any) containing the Software and/or Documentation, **"Product"** means collectively the Media, Software, and Documentation, and all Product updates subsequently provided to You, and **"You"** means the owner or lessee of the hardware on which the Software is installed and/or used. **"BMC"** means BMC Software Distribution, Inc. unless You are located in one of the following regions, in which case "BMC" refers to the following indicated BMC Software, Inc. subsidiary: (i) Europe, Middle East or Africa --BMC Software Distribution, B.V., (ii) Asia/Pacific -- BMC Software Asia Pacific Pte Ltd., (iii) Brazil -- BMC Software do Brazil, or (iv) Japan -- BMC Software K.K. **If You enter into a separate, written software license agreement signed by both You and BMC or your authorized BMC reseller granting to you the rights to install and use this Product, then the terms of that separate, signed agreement will apply and this License is void.**

FULL-USE LICENSE. Subject to these terms and payment of the applicable license fees, BMC grants You this non-exclusive License to install and use one copy of the Software for your internal use on the number(s) and type(s) of servers or workstations for which You have paid or agreed to pay to BMC or your BMC reseller the appropriate license fee. If your license fee entitles You only to a License having a limited term, then the duration of this License is limited to that term; otherwise this License is perpetual, subject to the termination provisions below.

TRIAL LICENSE. If You have not paid or agreed to pay to BMC or your BMC Reseller the appropriate license fees for a full use license, then, **NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENSE**: (i) this License consists of a non-exclusive evaluation license ("Trial License") to use the Product for a limited time ("Trial Period") only for evaluation; (ii) during the Trial Period, You may not use the Software for development, commercial, production, database management or other purposes than those expressly permitted in clause (i) immediately above; and (iii) your use of the Product is on an ***AS IS*** basis, and **BMC, ITS RESELLERS AND LICENSORS GRANT NO WARRANTIES OR CONDITIONS (INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE) TO YOU AND ACCEPT NO LIABILITY WHATSOEVER RESULTING FROM THE USE OF THIS PRODUCT UNDER THIS TRIAL LICENSE.** If You use this Product for other than evaluation purposes or wish to continue using it after the Trial Period, you must purchase a full-use license. When the Trial Period ends, your right to use this Product automatically expires, though in certain cases You may be able to extend the term of the Trial Period by request. Contact BMC or your BMC reseller for details.

TERM AND TERMINATION. This License takes effect on the first to occur of the date of shipment or accessibility to You for electronic delivery, as applicable (the **"Product Effective Date"**). You may terminate this License at any time for any reason by written notice to BMC or your BMC reseller. This License and your right to use the Product will terminate automatically with or without notice by BMC if You fail to comply with any material term of this License. Upon termination, You must erase or destroy all components of the Product including all copies of the Software, and stop using or accessing the Software. Provisions concerning Title and Copyright, Restrictions (or Restricted Rights, if You are a U.S. Government entity) or limiting BMC's liability or responsibility shall survive any such termination.

TITLE AND COPYRIGHT; RESTRICTIONS. All title and copyrights in and to the Product, including but not limited to all modifications thereto, are owned by BMC and/or its affiliates and licensors, and are protected by both United States copyright law and applicable international copyright treaties. You will not claim or assert title to or ownership of the Product. To the extent expressly permitted by applicable law or treaty notwithstanding this limitation, You may copy the Software only for backup or archival purposes, or as an essential step in utilizing the Software, but for no other purpose. You will not remove or alter any copyright or proprietary notice from

copies of the Product. You acknowledge that the Product contains valuable trade secrets of BMC and/or its affiliates and licensors. Except in accordance with the terms of this License, You agree (a) not to decompile, disassemble, reverse engineer or otherwise attempt to derive the Software's source code from object code except to the extent expressly permitted by applicable law or treaty despite this limitation; (b) not to sell, rent, lease, license, sublicense, display, modify, time share, outsource or otherwise transfer the Product to, or permit the use of this Product by, any third party; and (c) to use reasonable care and protection to prevent the unauthorized use, copying, publication or dissemination of the Product and BMC confidential information learned from your use of the Product. **You will not export or re-export any Product without both the written consent of BMC and the appropriate U.S. and/ or foreign government license(s) or license exception(s).** Any programs, utilities, modules or other software or documentation created, developed, modified or enhanced by or for You using this Product shall likewise be subject to these restrictions. BMC has the right to obtain injunctive relief against any actual or threatened violation of these restrictions, in addition to any other available remedies. Additional restrictions may apply to certain files, programs or data supplied by third parties and embedded in the Product; consult the Product installation instructions or Release Notes for details.

LIMITED WARRANTY AND CONDITION. If You have purchased a Full-Use License, BMC warrants that (i) the Media will be, under normal use, free from physical defects, and (ii) for a period of ninety (90) days from the Product Effective Date, the Product will perform in substantial accordance with the operating specifications contained in the Documentation that is most current at the Product Effective Date. BMC's entire liability and your exclusive remedy under this provision will be for BMC to use reasonable best efforts to remedy defects covered by this warranty and condition within a reasonable period of time or, at BMC's option, either to replace the defective Product or to refund the amount paid by You to license the use of the Product. BMC and its suppliers do not warrant that the Product will satisfy your requirements, that the operation of the Product will be uninterrupted or error free, or that all software defects can be corrected. This warranty and condition shall not apply if: (i) the Product is not used in accordance with BMC's instructions, (ii) a Product defect has been caused by any of your or a third party's malfunctioning equipment, (iii) any other cause within your control causes the Product to malfunction, or (iv) You have made modifications to the Product not expressly authorized in writing by BMC. No employee, agent or representative of BMC has authority to bind BMC to any oral representations, warranties or conditions concerning the Product. **THIS WARRANTY AND CONDITION IS IN LIEU OF ALL OTHER WARRANTIES AND CONDITIONS. THERE ARE NO OTHER EXPRESS OR IMPLIED WARRANTIES OR CONDITIONS, INCLUDING THOSE OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, REGARDING THIS LICENSE OR ANY PRODUCT LICENSED HEREUNDER. THIS PARAGRAPH SHALL NOT APPLY TO A TRIAL LICENSE.** Additional support and maintenance may be available for an additional charge; contact BMC or your BMC reseller for details.

LIMITATION OF LIABILITY. Except as stated in the next succeeding paragraph, BMC's and your BMC reseller's total liability for all damages in connection with this License is limited to the price paid for the License. **IN NO EVENT SHALL BMC BE LIABLE FOR ANY CONSEQUENTIAL, SPECIAL, INCIDENTAL, PUNITIVE OR INDIRECT DAMAGES OF ANY KIND ARISING OUT OF THE USE OF THIS PRODUCT (SUCH AS LOSS OF PROFITS, GOODWILL, BUSINESS, DATA OR COMPUTER TIME, OR THE COSTS OF RECREATING LOST DATA), EVEN IF BMC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.** Some jurisdictions do not permit the limitation of consequential damages so the above limitation may not apply.

INDEMNIFICATION FOR INFRINGEMENT. BMC will defend or settle, at its own expense, any claim against You by a third party asserting that your use of the Product within the scope of this License violates such third party's patent, copyright, trademark, trade secret or other proprietary rights, and will indemnify You against any damages finally awarded against You arising out of such claim. However, You must promptly notify BMC in writing after first receiving notice of any such claim, and BMC will have sole control of the defense of any action and all negotiations for its settlement or compromise, with your reasonable assistance. BMC will not be liable for any costs or expenditures incurred by You without BMC's prior written consent. If an order is obtained against your use of the Product by reason of any claimed infringement, or if in BMC's opinion the Product is likely to become the subject of such a claim, BMC will at its option and expense either (i) procure for You the right to continue using the product, or (ii) modify or replace the Product with a compatible, functionally equivalent, non-infringing Product, or (iii) if neither (i) nor (ii) is practicable, issue to You a pro-rata refund of your paid license fee(s) proportionate to the number of months remaining in the 36 month period following the Product Effective Date. This paragraph sets forth your only remedies and the total liability to You of BMC, its resellers and licensors arising out of such claims.

GENERAL. This License is the entire understanding between You and BMC concerning this License and may be modified only in a mutually signed writing between You and BMC. If any part of it is invalid or unenforceable, that part will be construed, limited, modified, or, severed so as to eliminate its invalidity or unenforceability. This License will be governed by and interpreted under the laws of the jurisdiction named below, without regard to conflicts of law principles, depending on which BMC Software, Inc. subsidiary is the party to this License: (i) BMC Software Distribution, Inc. - the State of Texas, U.S.A., (ii) BMC Software Distribution, B.V. - The Netherlands, (iii) BMC Software Asia Pacific Pte Ltd. -- Singapore (iv) BMC Software do Brazil -- Brazil, or (v) BMC Software K.K. -- Japan. Any person who accepts or signs changes to the terms of this License promises that they have read and understood these terms, that they have the authority to accept on your behalf and legally obligate You to this License. Under local law and treaties, the restrictions and limitations of this License may not apply to You; You may have other rights and remedies, and be subject to other restrictions and limitations.

U.S. GOVERNMENT RESTRICTED RIGHTS. UNPUBLISHED -- RIGHTS RESERVED UNDER THE COPYRIGHT LAWS OF THE UNITED STATES. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in FAR Section 52.227-14 Alt. III (g)(3), FAR Section 52.227-19, DFARS 252.227-7014 (b) or DFARS 227.7202, as amended from time to time. Contractor/Manufacturer is BMC Software, Inc., 2101 CityWest Blvd., Houston, TX 77042-2827, USA. Any contract notices should be sent to this address.

Notes

23355