

# **PATROL<sup>®</sup> Script Language Reference Manual Volume 1—PSL Essentials**

**Version 3.5**

**August 8, 2002**



Copyright 1994-2002 BMC Software, Inc., as an unpublished work.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 UNPUBLISHED—RIGHTS RESERVED UNDER THE COPYRIGHT LAWS OF THE UNITED STATES.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](http://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

---

# Contents

## About This Manual xiii

<b>Chapter 1</b>	<b>PATROL Script Language (PSL) Overview</b>	
	What Is PSL . . . . .	1-2
	Interpreted PSL Scripts . . . . .	1-2
	Compiled PSL Binary Files . . . . .	1-2
	Optimization . . . . .	1-2
	Diagnostics . . . . .	1-3
	PSL Built-in Functions . . . . .	1-3
	Locking Functions for Concurrency Control . . . . .	1-4
	Set Functions for PSL Lists . . . . .	1-4
	PSL Mathematical Functions . . . . .	1-5
	PSL Libraries . . . . .	1-6
	PSL Process Synchronization . . . . .	1-7
	PSL Shared Global Channels . . . . .	1-7
	Requirement for Shared Global Channels in PSL . . . . .	1-7
	Implementation of PSL Shared Global Channels . . . . .	1-8
	Effect of PSL Shared Global Channel Mechanisms . . . . .	1-9
	How to Use PSL in PATROL . . . . .	1-9
	Complex Application Discovery . . . . .	1-9
	Advanced User Commands . . . . .	1-10
	Efficient Monitoring Parameters . . . . .	1-10
	How PSL Relates to PATROL Architecture . . . . .	1-10
	Using Built-in or User-Defined Object Variables . . . . .	1-13
	PSL Naming Conventions . . . . .	1-13
<b>Chapter 2</b>	<b>PSL Data Types and Operators</b>	
	PSL Data Types and Objects . . . . .	2-2

Numeric Constants	2-2
PSL Variables	2-3
Default Initialization of PSL Variables	2-3
PSL Predefined Constants	2-4
PSL String Literals	2-5
PSL Here Documents	2-6
ActiveX Scripts	2-7
PSL Lists	2-8
PSL Simple Statements	2-8
PSL Operators	2-9
Arithmetic Operators	2-9
Assignment Operators	2-10
Increment/Decrement Operators	2-11
Bitwise Operators	2-11
Logical Operators	2-11
Relational Operators	2-12
Shift Operators	2-13
String Operators	2-13
Ternary Operator	2-14
PSL Operator Precedence and Associativity	2-15

## Chapter 3

### PSL Statements

Introduction	3-2
PSL Compound Statements	3-2
do...until	3-4
exit	3-6
export	3-7
for	3-9
foreach	3-11
function	3-13
return Statement	3-14
Functions with Variable Length Argument Lists	3-15
Defining Local Variables	3-16
Entry Point Function	3-17
Start of Execution Without an Entry Point Function	3-18
Backward Compatibility with Earlier PSL Versions	3-19
Limitations of User-Defined Functions	3-19
if	3-21
last	3-23
next	3-24

requires	3-25
switch	3-28
while	3-33

## Chapter 4

### PSL External Commands

%DUMP—List Specific Information	4-2
%DUMP CHANNELS—List PSL Global Channels	4-3
%DUMP LIBRARIES—List Loaded PSL Libraries	4-4
%PSL—Execute a PSL Statement	4-6
%PSLPS—List Current PSL Processes	4-7
psl—PSL Compiler Command	4-8

## Chapter 5

### Diagnosing PSL Program Errors

PsIDebug—Run-Time Error Checking Variable	5-2
errno—Error Return Code Variable	5-6
exit_status—System Return Code Variable	5-7
Incompatibilities with the C Programming Language	5-7
Operators && and	5-7
Prefix and Postfix Operators ++ and --	5-8
Break and Continue Statements	5-8
Common PSL Coding Errors	5-8
Character Strings Interpreted as Numbers	5-9
Floating Point Numbers Interpreted as Character Strings	5-10
Character Strings Interpreted as Variable Names	5-10
PSL Functions That Do Not Modify Their Arguments	5-11
Functions That Do Not Write to the Console Window	5-12
PSL Compiler Warnings	5-12
Built-in Function Run-Time Error Messages	5-15

## Chapter 6

### Internationalized PSL Scripts

Introduction	6-2
Locale and Codeset	6-2
Locale Categories	6-3
set_locale()	6-4
CTYPE Locale Category	6-5
MESSAGES Locale Category	6-5
CODECVT Locale Category	6-6
TIME Locale Category	6-7
Multiple-Byte Characters	6-7
PSL International Functions	6-7

ID-Based Messaging Functions .....	6-8
Other PSL Functions .....	6-9
Command Execution Functions .....	6-9
Input and Output Functions .....	6-10
File Handling Functions .....	6-11
String Functions .....	6-11
Set Functions .....	6-12
Date and Time .....	6-12
Compatibility with Noninternationalized PATROL Agents .....	6-13
Example Code: Verify the Version of the PATROL Agent .....	6-13
Example Code: Conditionally Use an International Function .....	6-15

**Appendix A      errno Return Values**

**Appendix B      Built-in Agent Namespace Variables**

Computer Class Built-in Variables .....	B-2
Application Class Built-in Variables .....	B-3
Application Instance Built-in Variables .....	B-4
Parameter Built-in Variables .....	B-5

**Appendix C      Additional PSL Tools**

PSL Profiler Tool .....	C-2
How to Install the PSL Profiler .....	C-4
How to Start the PSL Profiler .....	C-4
PSL Profiler PSL Functions .....	C-5
About the PSL Profile Viewer (ppv) Tool .....	C-8
About the PSL Profiler API .....	C-8
PSL Optimizer Tool .....	C-10
Introduction to the PSL Optimizer .....	C-10
How to Install the PSL Optimizer .....	C-10
How to Deactivate the PSL Optimizer .....	C-11
About the PSL Optimizer .....	C-11
Optimization Levels .....	C-11
Optimization Criteria .....	C-14
Command-Line Specified Options .....	C-16

**Index**

---

# Figures

Figure 1-1      Tree Structure of PATROL Applications and Objects . . . . . 1-11



---

## Tables

Table 1-1	Recommended Naming Conventions for PATROL Objects .	1-14
Table 2-1	Examples of PSL Data Types. . . . .	2-2
Table 2-2	PSL Predefined Constants . . . . .	2-4
Table 2-3	PSL String Literals. . . . .	2-6
Table 2-4	PSL Arithmetic Operators . . . . .	2-9
Table 2-5	PSL Assignment Operators . . . . .	2-10
Table 2-6	PSL Increment/Decrement Operators . . . . .	2-11
Table 2-7	PSL Bitwise Operators. . . . .	2-11
Table 2-8	PSL Logical Operators. . . . .	2-12
Table 2-9	PSL Relational Operators . . . . .	2-12
Table 2-10	PSL Shift Operators. . . . .	2-13
Table 2-11	PSL Operator Precedence and Associativity . . . . .	2-15
Table 5-1	PSlDebug Error Checking Flag Bits. . . . .	5-3
Table 6-1	Supported Locale Names . . . . .	6-2
Table 6-2	PSL Locale Categories. . . . .	6-4
Table 6-3	PSL International Functions . . . . .	6-8
Table 6-4	International Features of Command Execution Functions. . .	6-10
Table 6-5	International Features of Input and Output Functions. . . . .	6-10
Table 6-6	International Features of File Handling Functions . . . . .	6-11
Table 6-7	International Features of String Functions. . . . .	6-11
Table 6-8	International Features of Set Functions . . . . .	6-12
Table 6-9	International Features of Date and Time Functions. . . . .	6-12
Table A-1	PSL errno Values . . . . .	A-1
Table B-1	Computer Class Built-in Variables. . . . .	B-2
Table B-2	Application Class Built-in Variables . . . . .	B-3
Table B-3	Application Instance Built-in Variables. . . . .	B-4
Table B-4	Parameter Built-in Variables . . . . .	B-5



---

## About This Manual

This book introduces the PATROL<sup>®</sup> 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 1—PSL Essentials*, is the first volume of the PSL documentation, and it presents 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<sup>®</sup> Console and Knowledge Module<sup>™</sup> (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 information about PSL basics such as statement syntax and data types. If you need information about the built-in functions, refer to the *PATROL Script Language Reference Manual Volume 2—PSL Functions*.

## How This Manual Is Organized

The *PATROL Script Language Reference Manual Volume 1—PSL Essentials* is organized into the following chapters.

Chapter	Title	Purpose
1	"PATROL Script Language (PSL) Overview"	Introduces the PATROL Script Language (PSL), briefly explaining what PSL is and how it works.
2	"PSL Data Types and Operators"	Identifies the available PSL data types, constants, and operators.
3	"PSL Statements"	Describes the statements inherent in PSL. These statements include <ul style="list-style-type: none"><li>• <i>do...until</i>, <i>if</i>, <i>for</i>, <i>foreach</i>, and <i>while</i> for basic program loops</li><li>• <i>function</i> for defining user-defined functions</li><li>• <i>requires</i> and <i>export</i> for importing and exporting PSL libraries into a program</li></ul>
4	"PSL External Commands"	Describes commands that you enter from the operating system command line, including commands for the PSL compiler, the one-line PSL function entry, and the PSL process list.

Chapter	Title	Purpose
5	"Diagnosing PSL Program Errors"	Provides help in diagnosing PSL errors through debugging aids and error messages. This chapter describes: <ul style="list-style-type: none"> <li>• <i>PsIDebug</i>, <i>errno</i>, and <i>exit_status</i> variables</li> <li>• incompatibilities with the C programming language</li> <li>• common coding errors</li> <li>• compiler warnings</li> <li>• built-in function run-time error messages</li> </ul>
6	"Internationalized PSL Scripts"	Shows how to write internationalized PSL scripts, which have the following characteristics: <ul style="list-style-type: none"> <li>• Run on two or more computer platforms of different languages.</li> <li>• Read and write multilingual characters and formats such as date, time, and currency.</li> <li>• Exchange information between computer platforms of different languages.</li> </ul>
A	"errno Return Values"	Lists the error codes that are returned by the PSL built-in functions.
B	"Built-in Agent Namespace Variables"	The built-in Agent namespace variables maintained by the PATROL Agent are presented here in table format. You can use these variables to customize existing commands and parameters.
C	"Additional PSL Tools"	Describes additional PSL tools that can be used when developing PSL scripts.

## Related Documentation

BMC Software products offer several types of documentation:

- online and printed books
- online Help
- release notes

## 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>.

## 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:

- 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.

- 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.

## 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:

<b>Unix Button</b>	<b>Windows Button</b>	<b>Description</b>
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.

---



---

# PATROL Script Language (PSL) Overview

This chapter provides an overview of PSL. The following topics are discussed:

What Is PSL . . . . .	1-2
Interpreted PSL Scripts . . . . .	1-2
Compiled PSL Binary Files . . . . .	1-2
Optimization . . . . .	1-2
Diagnostics. . . . .	1-3
PSL Built-in Functions. . . . .	1-3
Locking Functions for Concurrency Control. . . . .	1-4
Set Functions for PSL Lists . . . . .	1-4
PSL Mathematical Functions . . . . .	1-5
PSL Libraries . . . . .	1-6
PSL Process Synchronization. . . . .	1-7
PSL Shared Global Channels . . . . .	1-7
Requirement for Shared Global Channels in PSL . . . . .	1-7
Implementation of PSL Shared Global Channels . . . . .	1-8
Effect of PSL Shared Global Channel Mechanisms . . . . .	1-9
How to Use PSL in PATROL . . . . .	1-9
Complex Application Discovery . . . . .	1-9
Advanced User Commands . . . . .	1-10
Efficient Monitoring Parameters . . . . .	1-10
How PSL Relates to PATROL Architecture . . . . .	1-10
Using Built-in or User-Defined Object Variables. . . . .	1-13
PSL Naming Conventions . . . . .	1-13

# What Is PSL

PSL is both an interpreted and a compiled language for writing complex application discovery procedures, parameters, and commands within the PATROL environment. It is also a good language for writing arbitrary commands and tasks.

PSL has been designed to provide functions needed to efficiently develop Knowledge Modules for the PATROL environment. To accomplish this, PSL sacrifices some of the completeness of languages such as C, csh, Perl, or awk, while implementing some of the statements and functions that make those languages so powerful and popular. Users familiar with one of those languages should have little difficulty adapting to PSL.

## Interpreted PSL Scripts

Upon receiving a script, PSL compiles the whole script into an internal form. If the script is syntactically correct, the internal form is then interpreted. Upon reexecution, the script is not recompiled.

## Compiled PSL Binary Files

PSL includes a standalone compiler to compile PSL scripts and create executable PSL binaries outside the PATROL<sup>®</sup> Agent. There is also an external interpreter that simulates all aspects of PSL execution except those areas specific to the PATROL Agent. See “psl—PSL Compiler Command” on page 4-8 for more information on compiling PSL scripts.

## Optimization

Like C, PSL does a certain amount of expression evaluation at compile time, whenever it determines that all of the arguments to an operator are static and have no side effects.

In particular, string concatenation is done at compile time between string literals. Backslash interpretation is also done at compile time. For example, the following expression is reduced to one string internally at compile time:

```
"Now is the time for all" . "\n" .  
"good men to come to."
```

PSL is also highly efficient. The language is not immediately interpreted, but instead is compiled into an internal form, which is then executed by a high-speed interpreter.

## Diagnostics

If any compilation errors are encountered when a PSL script is compiled, error messages tell you the line number of the error and briefly describe the cause of the error. See Chapter 4, “PSL External Commands,” for more information.

## PSL Built-in Functions

PSL includes a number of built-in functions that provide PATROL-specific actions such as creating and manipulating PATROL objects and general-purpose functions such as mathematical, logical, and I/O functions. The functions are individually described in the *PATROL Script Language Reference Manual Volume 2—PSL Functions*. The following sections summarize the PSL built-in functions.

# Locking Functions for Concurrency Control

PSL includes the `lock()` and `unlock()` built-in functions for enforcing concurrency control. These functions are typically used to linearize accesses by different PSL parameters, commands, and processes to shared data structures. These shared data structures include the object hierarchy accessed by the PSL `set()` and `get()` functions and external resources such as files.

All PSL processes attempting to linearize accesses to a resource must cooperate by requesting locks of a given lock name. All resource accesses, including the `set()` and `get()` functions, are denied shared resource access without a lock. It is the responsibility of each PSL process to access a resource only when it holds the required lock.

## Set Functions for PSL Lists

PSL includes the following functions for performing set operations on PSL lists:

- `difference()`—returns the list of different elements between lists
- `intersection()`—returns a list of elements common between lists
- `sort()`—returns a list in ascending or descending element order
- `subset()`—verifies that one list is contained within another
- `union()`—returns a list that is a combination of lists
- `unique()`—returns a list of elements that appears in only one list

These functions process PSL lists as sets of elements. Each member of a list is text string that ends with a new-line character. (The text string cannot contain embedded new-line characters.) Although the set functions will properly handle lists whose last element does not end with a new-line character, BMC Software recommends that you ensure that all elements, including the last element, are followed by a new-line character.

The NULL set [ " " ] is the equivalent of the null or empty set ( $\phi$ ) in set theory. The NULL set is treated by the PSL set functions as a proper set that contains no elements.

The NULL string [ ] is a PSL list element with no characters. The PSL set functions allow lists to contain NULL strings, but BMC Software discourages their use because their unique characteristics can produce unexpected results.

The PSL concept of a set is not the unique list of ascending or descending elements familiar to set theory. In many cases, the PSL lists contain duplicate elements arranged in no particular order. A PSL list can be transformed into an ordered set using the `unique()` function to remove duplicates and the `sort()` function to arrange the elements in ascending or descending order.

## PSL Mathematical Functions

PSL supports a basic subset of mathematical functions. These functions are all based on the standard C mathematical functions and include the peculiarities those functions display in the standard C libraries on any given platform.

---

### Note

---

Although most issues involving the C mathematical functions are standardized, there are platform-specific differences such as levels of accuracy; the different handling of various out-of-range conditions may cause them to behave differently on different platforms.

---

All PSL functions except one have the same name as the corresponding C function. The exception is that the `loge()` function corresponds to C's `log ( )` function because PSL already has a `log()` function used for logging events. The PSL `log()` function has been augmented to return a run-time error when it receives a numeric value. This error message reduces the likelihood of accidentally calling it when the `loge()` function is intended.

---

### Note

---

The `log()` function has been replaced by the `event_trigger()` function, but the `log()` function is still supported for backward compatibility.

---

The PSL mathematical functions include some run-time error checking for range and domain. Both conditions result in a run-time error message that sets the PSL `errno` variable to an appropriate value.

Additionally, any nonnumeric values produced by printing the result of the function call, such as `NaN` or `-Inf`, are converted to `0.0` to prevent the return value from being interpreted by PSL as a nonnumeric character string. A PSL function also returns a run-time error message when it performs the conversion.

Note that some PSL domain errors may not exactly match those of the standard C library:

- The PSL functions and argument ranges raise a domain error and return zero.
- The C functions may raise a range error and/or return a nonnumeric representation of the result such as infinity in the following cases:
  - `loge(x)` and `log10(x)` when  $x \leq 0$
  - `sqrt(x)` when  $x < 0$
  - `fmod(x, 0)`

## PSL Libraries

PSL supports shared libraries of PSL user-defined functions. PSL libraries are binary files that can be loaded into another PSL program using the PSL `requires` statement. The PSL libraries can be loaded using the following methods:

- statically—mixing the library object code and symbol table into the PSL program to create a larger binary
- dynamically—requesting at PSL program startup that the PatrolAgent find the required library in its repository of libraries (that is, probably in the Knowledge Module or in the directory specified by the `PATROL_HOME` or `HOME` environment variable)

If the PATROL Agent cannot find the library, the program execution fails.

If the PATROL Agent finds the library, it dynamically loads it as a shared library. Shared libraries are loaded only once by the PATROL Agent; all PSL processes using that library (with dynamic loading) share the library object code and constants but have their own copy of library variables.

## PSL Process Synchronization

PSL provides process synchronization within PSL processes of a single PATROL Agent through condition variable primitives used with PSL locks. These primitives are similar to constructs provided for multi-threaded programming in the C programming language on many non-threading operating systems.

## PSL Shared Global Channels

PSL supports the use of shared global channels for communication between a process and another process or file.

### Requirement for Shared Global Channels in PSL

In previous versions of PSL, channels were restricted to a single PSL process. Restricting a channel to a single process meant that each PSL process that wished to communicate with, or query to, an external process such as a database had to open its own channel. Opening a new channel for each caller heavily increased resource usage.

Another problem with the previous channel mechanism was that there was no way for one PSL process to open the channel and another PSL process to use it. The usual workaround was to do everything in one collector parameter that performed all queries and set the consumer parameter values. That method had the following limitations:

- no way to have distinct poll times for the various consumers
- no convenient way to deactivate queries that were deemed unnecessary

All the agent scheduling features were lost unless the Knowledge Module developer could explicitly code the equivalent functionality into the collector's PSL code.

## Implementation of PSL Shared Global Channels

The current version of PSL allows one PSL parameter to open a channel to an external process in an explicitly shared mode, which allows any number of other PSL processes to send data to, and receive data from, the channel.

The ability to share channels also requires PSL to provide a mechanism for concurrent programming techniques. PSL provides these in the form of external synchronization primitives.

The primitives are preferable to building concurrency into the channel opening, reading, writing, and closing functions. For example, having each PSL process lock a shared channel explicitly prevents concurrent reading by one process and writing by another. In addition, having the synchronization primitives separate from channels allows them to be used to synchronize the use of any shared resource such as the agent's internal symbol table or an external file.

The current implementation of the `read()`, `readln()`, and `write()` functions for shared channels will fail immediately (without blocking) if another PSL process is already blocked on the channel. The Knowledge Module developer must use PSL locks and/or PSL condition variables to enforce the desired synchronization of the use of shared channels.

## Effect of PSL Shared Global Channel Mechanisms

The PSL functions ensure that all operations on a channel are serialized, with all PSL function calls appearing to be atomic. The PSL programmer can be assured that file channel reads and writes in different processes will take place atomically. The locks provided in PSL prevent unpredictable interleaving of sequences of PSL read and write calls to the channel.

The single exception to serialization on channels created using the `popen()` function is the allowance for a concurrent read and write operation. A read can occur when a write is pending on the channel, and a write can occur when a read is blocked or pending on the channel—thus, both a reader and a writer PSL process can be blocked on a shared channel.

File channels opened using the `fopen()` function can never cause a PSL `read()` or `write()` function to block. To enforce serialization, the second reader process cannot be blocked, nor can the second writer process be blocked; hence, the second PSL `read()`, `readln()`, and `write()` functions on a file channel will fail.

## How to Use PSL in PATROL

Knowledge Module developers and sophisticated users use PSL to extend the PATROL management environment. The PSL scripts that you create are sent to the appropriate `PatrolAgents` where they are stored and utilized.

The following topics briefly describe typical uses for PSL scripts.

### Complex Application Discovery

You could write a PSL script (called a *discovery script*) to identify all available printers and their status. The PSL script could then cause appropriate icons to be displayed by the PATROL Console.

Typical PSL discovery scripts include RDBMS discovery, file system discovery, and so on.

## Advanced User Commands

PATROL provides menus for all managed objects. PSL can be used to allow the stored commands on these menus to perform complex tasks. For example, a typical PSL command on a computer icon would produce a report on “CPU hogs” or allow the administrator to add a user.

## Efficient Monitoring Parameters

PSL parameters can analyze and manipulate host information generating a minimum of extra processes on the managed computer.

For example, you can use a PSL parameter to run a command such as the Unix `sar` command, which returns a multitude of data; then break this data down into many different values, which can be passed on to other parameters to be displayed. The parameter that gathers the data is called a *collector*, and the parameters that display the values are called *consumers*.

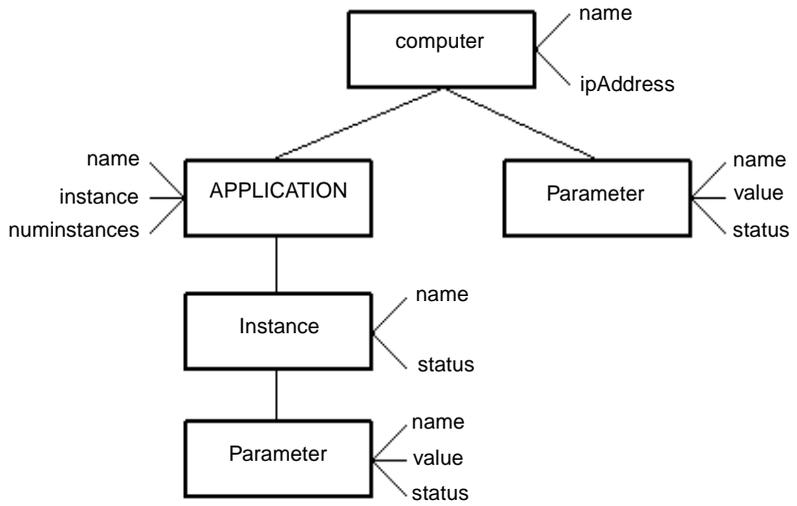
By using a collector parameter to run the `sar` command just once, collect its output, and then pass the many values to the various consumer parameters, you can avoid executing the `sar` command numerous times to get each parameter value, thereby conserving valuable operating system resources.

## How PSL Relates to PATROL Architecture

Managing objects in the PATROL environment requires an understanding of the naming conventions PATROL uses for objects, instances, parameters, and variables. See “PSL Naming Conventions” on page 1-13. You can access objects using their absolute name or their name relative to the PSL programming context.

Figure 1-1 shows how application objects can have variables and instances. In turn, the application instances can have variables and parameters, which in turn can have variables.

**Figure 1-1 Tree Structure of PATROL Applications and Objects**



As shown in Figure 1-1, the root of the hierarchy is the computer. The computer (root) directory contains the applications. Each application directory contains all the application instances. Each instance directory contains all the parameters for each instance and so on.

The designation **/RDB/Dev** refers to the RDB database named **Dev**. Variables (such as “name”) are analogous to files within a directory. The status of this database instance would be referenced as **/RDB/Dev/status**.

Conceivably, *status* could be the name of a parameter on Dev, in which case a conflict would exist. It is recommended that you begin all variable names with a lowercase letter and all object names (parameters, applications, etc.) with an uppercase letter to avoid name conflicts.

---

**Note**

---

Refer to “PSL Naming Conventions” on page 1-13 for more information on developing Knowledge Modules which conform to PSL style guidelines.

---

The designation **/RDB/Dev/Status** is an object (a parameter) because it begins with a capital letter and **/RDB/Dev/status** is a variable for the **Dev** instance of the **RDB** application.

The value of the parameter Logons on database Dev would be referenced as **/RDB/Dev/Logons/value**.

Once you define an object, you do not normally have to give the absolute name each time you reference it. You can make use of the context in which the script you are developing will run to construct shorter, relative names for most objects.

The context of a script is the name of the object to which the executing script belongs. For example, if you write a script for the discovery of the RDB application, its context will be the application class it is discovering.

Whenever you give the name of an object without a preceding “/”, the PatrolAgent prefixes the name with the current context. Using the previous example, if we gave the object name **Dev**, it would be expanded to **/RDB/Dev** automatically.

The reference “..” is reserved to mean the parent of the current context. For example, the reference “..**name**” in an application discovery script refers to the name of the computer since the computer is the parent object of all application class objects. In a parameter, it refers to the application instance *name* since the parent of a parameter is its application instance.

# Using Built-in or User-Defined Object Variables

All objects—computers, applications, classes, application instances, parameters—have variables. Variables can be either built-in (defined automatically by PATROL when the object is created) or defined by the user.

Built-in variables are used by PATROL to record information, such as the name and status of an object. The values of some built-in variables are static, whereas others are updated dynamically by the PatrolAgent. Most built-in variables cannot be modified—that is, they are read-only—but some can be changed by the user to control the behavior of the PatrolAgent.

For example, the *objectID* built-in variable (the unique internal object id) is read-only, whereas the *value* built-in variable (the value of a parameter) is read-write.

A user script can change the label of the instance's icon by setting the value of the *name* for the application instance variable:

```
set ( "/RDB/Dev/name" , "Development" );
```

## PSL Naming Conventions

To make it easier for you to determine whether a particular name describes an application class, an application instance, a parameter, or a variable, BMC Software recommends that you follow these naming conventions for PATROL objects:

**Table 1-1 Recommended Naming Conventions for PATROL Objects**

<b>PATROL Object Type</b>	<b>Convention</b>	<b>Example</b>
APPLICATION CLASS	Use all uppercase letters.	<b>/FILESYSTEM /MYAPP</b>
Application Instance	Use Initial capitalization. <sup>†</sup>	<b>/MYAPP/Instance1</b>
Parameter	Use a few letters of the parameter's application class. Initial cap each word in the name.	<b>/FILESYSTEM/etc/FSInodesPctUsed /MYAPP/Instance1/MYFirstParam</b>
variable	Begin with a lowercase letter. Initial cap each word in the name after the first word.	<b>/MYAPP/Instance1/numUsers</b>

<sup>†</sup>Generally, the instance's name is dictated by the name of the real-life object that it represents, so following this convention is not always possible, as in **/FILESYSTEM/etc**.

The object and variable naming conventions are similar to those of the Unix file system. They are *not* file names, and you should not be concerned if they do not conform to the file naming conventions for your particular operating system.

---

**Warning**

---

While naming objects and variables, avoid using special characters such as |, \*, &, \$, and @. The PATROL Agent reserves many special characters for specific uses. The | symbol, for example, functions as a code separator, and using it in an object name yields undesirable results.

---

---

# PSL Data Types and Operators

This chapter provides information on the syntax of PSL. The following topics are discussed:

PSL Data Types and Objects . . . . .	2-2
Numeric Constants . . . . .	2-2
PSL Variables . . . . .	2-3
Default Initialization of PSL Variables . . . . .	2-3
PSL Predefined Constants . . . . .	2-4
PSL String Literals . . . . .	2-5
PSL Here Documents . . . . .	2-6
ActiveX Scripts . . . . .	2-7
PSL Lists . . . . .	2-8
PSL Simple Statements . . . . .	2-8
PSL Operators . . . . .	2-9
Arithmetic Operators . . . . .	2-9
Assignment Operators . . . . .	2-10
Increment/Decrement Operators . . . . .	2-11
Bitwise Operators . . . . .	2-11
Logical Operators . . . . .	2-11
Relational Operators . . . . .	2-12
Shift Operators . . . . .	2-13
String Operators . . . . .	2-13
Ternary Operator . . . . .	2-14
PSL Operator Precedence and Associativity . . . . .	2-15

# PSL Data Types and Objects

PSL has four data types: integer, float, string, and list; however, all four types are represented internally as character strings. Table 2-1 lists the PSL data types:

**Table 2-1 Examples of PSL Data Types**

Data Type	Example	Representation
integer	3	" 3 "
float	4.5	" 4 . 5 "
string	"abc"	" abc "
list	[1,3,5]	" 1 \n 3 \n 5 "

Note: "\n" is the new-line character.

Variables and values are interpreted as either strings or numbers, whichever is appropriate to the context.

A scalar (integer or float) is interpreted as true in the Boolean sense if it is not the null string or 0. Booleans returned by operators are 1 for true and 0 or "" (the null string) for false.

## Numeric Constants

Although the internal representation of an integer or floating-point constant is a string, these constants do not need to appear inside quotation marks in PSL scripts. Integer and floating-point constants can be used in the same way as in the C programming language:

```
x = 3;  
pi = 3.14159;
```

## PSL Variables

Variables of any type can be used as values—that is, they can be assigned to. As all data types are treated as strings internally, they all share a common name space. Therefore, you cannot use the same name for a scalar variable, a string variable, and a list variable.

Case is significant. FOO, Foo, and foo are all different names. Names must start with a letter or an underscore but can contain digits and underscores (“\_”).

Some identifiers have predefined meanings. Reserved keywords—such as *if* and *foreach*—cannot be used as identifiers. Keywords are recognized as either all lowercase or all uppercase letters. In addition, the predefined constants listed in Table 2-2, “PSL Predefined Constants,” on page 2-4 cannot be used as identifiers.

## Default Initialization of PSL Variables

PSL does not make use of the concept of “declarations” for variables. The first appearance of an identifier serves to add it to the list of global variables for a PSL script. All variables are initialized with a null string value each time a PSL script is executed. This value does not change until the variable’s value is defined by some explicit operation, such as assignment.

This default initialization to the null string allows a variable to be treated as an initially empty list/string or as a numeric variable with a 0 value (since arithmetic operators treat the null string as equivalent to 0). However, reliance on this initial value causes a PSL run-time warning message at its first use (if run-time warnings are enabled). It is considered better style to initially assign a value of “” or 0 to a list/string variable or numeric variable, respectively.

---

### Note

For more information on enabling run-time warnings, see “PslDebug—Run-Time Error Checking Variable” on page 5-2.

---

## PSL Predefined Constants

A number of identifiers are predefined as constants so that they can be used without needing declaration. The predefined constants are used as PSL function parameters, PATROL object states, and for other PSL processing. These constants are read-only and will not accept user defined values. Table 2-2 lists the PSL predefined constants:

**Table 2-2 PSL Predefined Constants (Part 1 of 2)**

Constant	Definition
<b>chart() Function Actions</b>	
CHART_ADD_GRAPH	not currently implemented
CHART_DELETE_GRAPH	not currently implemented
CHART_DESTROY	destroys a previously loaded chart
CHART_LOAD	loads a chart
CHART_PRINT	prints a previously loaded chart
<b>PATROL Object States</b>	
ALARM	PATROL ALARM object state
WARN	PATROL WARNING object state
OK	PATROL OK object state
OFFLINE	PATROL OFFLINE object state
VOID	PATROL VOID object state
<b>response() Function Elements</b>	
R_CHECK_HORIZ	horizontal check box
R_CHECK_VERT	vertical check box
R_CLICKER	clicker widget
R_COLUMN	column compound
R_FRAME	frame compound
R_ICON	icon
R_LABEL	left-justified label
R_LABEL_CENTER	centered label
R_LIST_MULTIPLE	multiple-select scrolled list with defaults

**Table 2-2 PSL Predefined Constants (Part 2 of 2)**

<b>Constant</b>	<b>Definition</b>
R_LIST_MULTIPLE_ND	multiple-select scrolled list without defaults
R_LIST_SINGLE	single-select scrolled list with defaults
R_LIST_SINGLE_ND	single-select scrolled list without defaults
R_MENU	option menu
R_POPUP	non scrolled pop-up
R_POPUP_SCROLLED	scrolled pop-up
R_RADIO_HORIZ	horizontal radio button
R_RADIO_VERT	vertical radio button
R_ROW	row compound element
R_SCALE_HORIZ	horizontal sliding scale
R_SCALE_VERT	vertical sliding scale
R_SEP_HORIZ	horizontal separator
R_SEP_VERT	vertical separator
R_SPINNER	time spinner button
R_TEXT_FIELD	text entry box without a label
R_TEXT_FIELD_LABEL	text entry box with a label
R_TOGGLE	toggle button
<b>Other Constants</b>	
EOF	end-of-file condition constant
true/TRUE/True yes/YES/Yes	boolean true value (logical 1)
false/FALSE/False no/NO/No	boolean false value (logical 0)

## PSL String Literals

String literals are delimited by double quotation marks. String literals can be multiline, causing the new-line characters to become part of the string.

The backslash rules apply for escaping characters (such as the backslash or the quotation mark) and for making characters such as new-line or tab. Table 2-3 list the string literals currently supported in PSL:

**Table 2-3 PSL String Literals**

<b>Constant</b>	<b>Definition</b>
<code>\t</code>	tab
<code>\n</code>	new-line
<code>\r</code>	return
<code>\b</code>	backspace
<code>\A . . . \Z</code>	Ctrl-A . . . Ctrl-Z

Control characters can be embedded in PSL string constants using `\A` through `\Z` to represent `Ctrl-A` through to `Ctrl-Z`. A capitalized letter must always be used; lowercase letters other than those already defined (that is, `t`, `n`, `r`, or `b`) are not valid as escapes and will generate a PSL compilation warning.

When using a `\` (backslash) in a string literal, two `\\` (back slashes) must be used because the first will be interpreted as a control character, and the second will be used as part of the string.

A path for a Windows host is represented in a PSL string as follows:

```
C:\\PATROL\\lib\\PSL\\
```

The above string will be interpreted as the following:

```
C:\PATROL\lib\PSL\
```

## PSL Here Documents

A "here document" is a free-form string which is not modified by the PSL compiler. Special characters like `\n` (new line) and `\t` (tab), which are used to control output format in strings, are not interpreted when they are contained in here document constructions.

The here document construction starts with <<< followed by a string that will be used as the here document text delimiter. The here document text string continues until the delimiter is repeated at the beginning of a line. The here document delimiter can be any string consisting of letters, numerals, and underscores (\_), without internal blanks.

The here document construction can be used anywhere that a variable or string can be used, or it can be assigned to a variable.

### Example

```
here_document= <<<Here_Doc_DELIMIT
This is a free-form string.
Special characters like \n are not translated.
However, they would be in a quoted string.
This here document will continue until the delimiter
'Here_Doc_DELIMIT' is found at the beginning of a line.
That means the above 'Here_Doc_DELIMIT' was not the end.
The next line is the end delimiter, not part of the string.
Here_Doc_DELIMIT;
print(here_document);
```

This example returns the following output:

```
This is a free-form string.
Special characters like \n are not translated.
However, they would be in a quoted string.
This here_document will continue until the delimiter
'Here_Doc_DELIMIT' is found at the beginning of a line.
That means the above 'Here_Doc_DELIMIT' was not the end.
The next line is the end delimiter, not part of the string.
```

Everything between the two delimiters, Here\_Doc\_DELIMIT, represents the here\_document variable text string.

## ActiveX Scripts

With the PATROL Agent for Windows, the execute() function can submit an ActiveX script to the PATROL Scripting Host. The ActiveX script can be VBScript, JScript, or any valid Microsoft ActiveX scripting language.

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.

---

**Note**

---

The ActiveX script option is only available with the PATROL Agent for Windows.

---

For more information on submitting ActiveX scripts with the `execute()` function, see the *PATROL Script Language Reference Manual Volume 2—PSL Functions*.

## PSL Lists

List values are denoted by separating individual values with commas and by enclosing the list in square brackets:

```
[1, 3, 5]
```

The list is interpolated into a double-quoted string whose elements are separated by new-line characters. The list is represented internally as

```
"1\n3\n5"
```

## PSL Simple Statements

The most common simple statement is an expression evaluated for its side effects, which is called an expression statement. The most common expression statement is an assignment operation or a function call. Every expression statement must be terminated with a semicolon:

```
y = x + 10;          # assignment
set("value",50);    # function call
s = trim(s,"\t");   # both assignment and function call
```

PSL expression statements work like C expressions. The only difference is the addition of several string operators. See “String Operators” on page 2-13.

## PSL Operators

This section describes the operators available in PSL. PSL provides the following types of operators:

- arithmetic
- assignment
- increment and decrement
- bitwise
- logical
- relational
- shift
- string
- ternary

## Arithmetic Operators

For arithmetic operators, an operand is considered a number if its first character is a digit or a minus sign (–). Otherwise, it is considered a string and converted to 0 for an empty string or 1 for a nonempty string.

The use of a nonnumber in an arithmetic context may result in a run-time warning, as discussed in “PslDebug—Run-Time Error Checking Variable” on page 5-2. Table 2-4 lists the PSL arithmetic operators:

**Table 2-4 PSL Arithmetic Operators (Part 1 of 2)**

<b>Operator</b>	<b>Definition</b>
+	addition
-	subtraction
/	division

**Table 2-4 PSL Arithmetic Operators (Part 2 of 2)**

<b>Operator</b>	<b>Definition</b>
*	multiplication
%	modulus

## Assignment Operators

Table 2-5 lists the assignment operators for PSL (such as `a+=b` is equivalent to `a=a+b`):

**Table 2-5 PSL Assignment Operators**

<b>Operator</b>	<b>Definition</b>
=	assignment
+=	self-addition
-=	self-subtraction
/=	self-division
*=	self-multiplication
%=	self-modulus
<b>Bitwise Assignment:</b>	
&=	self-bitwise AND
=	self-bitwise OR
^=	exclusive OR bitwise assignment
<b>Shift Assignment:</b>	
<<=	shift left assignment
>>=	shift right assignment

## Increment/Decrement Operators

For information about limitations with increment/decrement operators, see “Incompatibilities with the C Programming Language” on page 5-7. Table 2-6 lists the PSL increment and decrement operators (such as `a++` is equivalent to `a=a+1`):

**Table 2-6 PSL Increment/Decrement Operators**

Operator	Definition
<code>++</code>	increment
<code>--</code>	decrement

## Bitwise Operators

Table 2-7 lists the bitwise operators defined for PSL (such as `a&=b` is equivalent to `a=a&b`):

**Table 2-7 PSL Bitwise Operators**

Operator	Definition
<code>&amp;</code>	bitwise AND
<code> </code>	bitwise OR
<code>&amp;=</code>	self-bitwise AND
<code> =</code>	self-bitwise OR
<code>^</code>	exclusive OR bitwise
<code>^=</code>	exclusive OR bitwise assignment

## Logical Operators

The PSL logical operators assume for their operands that true is represented by 1 or a nonempty string. False is represented by 0 or an empty string. However, when PSL logical operators return results, they always use 1 for true and 0 for false.

For further information about limitations with logical operators, see “Incompatibilities with the C Programming Language” on page 5-7. Table 2-8 lists the PSL logical operators:

**Table 2-8 PSL Logical Operators**

<b>Operator</b>	<b>Definition</b>
&&	logical AND
	logical OR
!	logical negation (NOT)

## Relational Operators

The relational operators perform numeric comparisons if both operands are numbers. Otherwise they perform string comparisons (that is, lexical, dictionary ordering).

A string is considered a number if it consists of only digits, the minus sign, or a period. No white space is allowed. PSL relational operators do not consider constants in C-like exponential notation (such as  $2.3e+27$ ) to be numbers. Table 2-9 lists the PSL relational operators:

**Table 2-9 PSL Relational Operators**

<b>Operator</b>	<b>Description</b>
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

## Shift Operators

Shift operators perform bit shifting within bytes. Table 2-10 lists the PSL shift operators:

**Table 2-10 PSL Shift Operators**

Operator	Description
<<	shift left
<<=	shift left assignment
>>	shift right
>>=	shift right assignment

## String Operators

PSL has special operators for string and list manipulation that are not found in C.

### . (period)

The period indicates the concatenation of two strings.

"ab" . "cd" is equal to "abcd"

### [s1, s2, ...]

The list operator builds a list by joining all elements in a comma-separated list into a double-quoted string of items delimited by a new-line characters, which is PSL's representation for lists/arrays. [ "a" , "b" , "c" ] is equal to "a\nb\nc"

---

#### 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.

---

## **=~ (equal tilde)**

The `=~` operator is used in the expression `string =~ pattern` and returns the following values:

- 1 if the regular expression *pattern* is contained in *string*
- 0 if the regular expression *pattern* is not contained in *string*

If *pattern* is invalid, PSL returns a run-time error message, and the `=~` operation returns 0 (*pattern* not contained).

## **!~ (tilde)**

The `!~` operator is used in the expression `string !~ pattern` and returns the following values:

- 1 if the regular expression *pattern* is not contained in *string*
- 0 if the regular expression *pattern* is contained in *string*

If *pattern* is invalid, PSL returns a run-time error message and the `!~` operation returns 0 (*pattern* contained).

## **Ternary Operator**

The PSL ternary operator `?:` behaves similarly to the C conditional expression in that it connects three operands and offers an alternative way of expressing a simple `if...else` statement. The format for the PSL conditional expression is as follows:

```
result = expression1 ? expression2 : expression3;
```

If *expression<sub>1</sub>* is TRUE (nonzero), then *expression<sub>2</sub>* is evaluated; otherwise, *expression<sub>3</sub>* is evaluated. The value for the complete conditional expression is the value of either *expression<sub>2</sub>* or *expression<sub>3</sub>*, depending on which expression was evaluated. The value of the expression may be assigned to a variable.

Conditional expressions are most useful in replacing short, simple `if...else` statements. For example, the `if...else` statement

```

if (x==1) {
    y=10;
}
else {
    y=20;
}

```

can be replaced with the one-line conditional statement

```
y = (x==1) ? 10 : 20;
```

These examples perform the same function. If x is 1, then y becomes 10 else y becomes 20.

## PSL Operator Precedence and Associativity

The precedence and associativity of PSL operators is almost identical to that of C and Perl. Table 2-11 lists the PSL operators in ascending order of precedence:

**Table 2-11 PSL Operator Precedence and Associativity (Part 1 of 2)**

Operator	Precedence	Associativity
=	lowest	right
+=, -=, <<=, >>=, ^=		right
*=, /=, %=		right
=, &=		right
?: (ternary)		right
		left
&&		left
		left
^		left
&		left
!=, ==, =~, !~		left
<, <=, >, >=		left
<<, >>		left

**Table 2-11 PSL Operator Precedence and Associativity (Part 2 of 2)**

<b>Operator</b>	<b>Precedence</b>	<b>Associativity</b>
+, - (binary)		left
*, /, %		left
. (string concat)		left
-, !, ++, --		right
()		left
[]	highest	left

---

# PSL Statements

This chapter describes the statements that are supported by PSL. The following topics are discussed:

Introduction .....	3-2
PSL Compound Statements .....	3-2
do...until .....	3-4
exit .....	3-6
export .....	3-7
for .....	3-9
foreach .....	3-11
function.....	3-13
return Statement.....	3-14
Functions with Variable Length Argument Lists.....	3-15
Defining Local Variables .....	3-16
Entry Point Function .....	3-17
Start of Execution Without an Entry Point Function.....	3-18
Backward Compatibility with Earlier PSL Versions .....	3-19
Limitations of User-Defined Functions .....	3-19
if .....	3-21
last.....	3-23
next .....	3-24
requires.....	3-25
switch .....	3-28
while .....	3-33

# Introduction

A PSL script consists of a sequence of commands. All uninitialized user-created objects are assumed to start with a NULL or 0 value until they are defined by some explicit operation such as assignment.

PSL is, for the most part, a free-form language. That is, lines don't have to start or end at or before a particular column; they can just continue on the next line. White space is ignored except for the separation of tokens. Comments are indicated by the # character and extend to the end of the line.

For example, here is a comment about an assignment statement:

```
x = y;    # Assign the value of y to the variable x
```

## PSL Compound Statements

PSL compound statements include loop statements and if statements. In PSL, a sequence of statements can be treated as one statement by enclosing it in braces {}. We will call this a statement block and denote it in the statement descriptions as {*BLOCK*}. The following compound statements can be used to alter or change the flow of control in a PSL program:

```
if (expression) {BLOCK}

if (expression) {BLOCK} else {BLOCK}

if (expression) {BLOCK}
elseif (expression) {BLOCK}
. . .
else {BLOCK}

foreach variable (array) {BLOCK}

foreach unit variable (array) {BLOCK}
```

```
switch(variable)
{
    case m: {BLOCK}
        . . .
    case n: {BLOCK}
    default: {BLOCK}
}

while (expression) {BLOCK}

do {BLOCK} until (expression);

for ([initexpr]; [termexpr]; [reinitexpr]) {BLOCK}
```

---

**Note**

---

These compound statements are defined in terms of statement blocks, not statements. This means that the braces are *required* rather than optional. No dangling statements are allowed.

---

# do...until

Loops through a *BLOCK* of PSL code until an *expression* is evaluated as TRUE

## Format

```
do {  
    {BLOCK}  
} until (expression);
```

## Parameters

Parameter	Definition
BLOCK	one or more PSL statements that are repeatedly executed until the evaluation of <i>expression</i> is TRUE
expression	a PSL statement whose evaluation returns either TRUE or FALSE  If TRUE, the loop terminates.

## Description

The PSL do...until loop behaves similarly to the C do...while loop construct in that it tests the termination condition at the end of the loop after making each pass through the body of the loop. Therefore, the body is always executed at least once. At first pass, the *BLOCK* of statements is executed, then *expression* is evaluated. If *expression* is FALSE, the *BLOCK* of statements is executed again. Iteration repeats until *expression* is TRUE.

## Example

The following example demonstrates the PSL do...until loop:

```
i = 10;
do {
    printf(" %d seconds to go\n",i);
    i--;
    sleep(i);
} until ( i == 0);
```

This example produces the following output:

```
10 seconds to go
 9 seconds to go
 8 seconds to go
 7 seconds to go
 6 seconds to go
 5 seconds to go
 4 seconds to go
 3 seconds to go
 2 seconds to go
 1 seconds to go
```

# exit

Immediately terminates the execution of a PSL program

## Format

```
exit;
```

## Parameters

This statement has no parameters.

## Description

The exit statement causes the PSL program to immediately end and return control to the process that called it. The exit statement must be terminated with a semicolon when used in a PSL program.

# export

Makes a variable or a function in a PSL library available to other PSL libraries and functions

## Format

```
export variable;  
export function function;
```

## Parameters

Parameter	Definition
variable	name of a PSL variable that is available for export to another PSL program
function	name of a PSL function that is available for export to another PSL program

## Description

The `export` statement makes a variable or function in a PSL library available for export to another PSL library or program using the `requires` statement. Each export statement can specify a single variable or function.

Global variables and functions need not be declared before the `export` statement. The `export` statement does not require that a variable be explicitly defined within a library, but it does require that it appear in a PSL statement to create an implicit definition.

## Placement of the export Statement

The `export function function` statement can appear before or after the actual function definition. The `export variable` statement can appear before or after the first appearance of a global variable.

An export statement can appear inside a function definition without any special significance. BMC Software discourages placing export statements inside function definitions.

## Errors Involving the export Statement

The export statement can generate compiler errors in the following instances:

- *variable* or *function* is not defined or used in the library
- *variable* or *function* is a PSL built-in function
- *variable* is a local variable of a user-defined function in the library
- *variable* or *function* is duplicated in another export statement
- *variable* or *function* has been imported using the requires statement

# for

Loops through a BLOCK of PSL code for a specified period

## Format

```
for ([initexpr]; [termexpr]; [reinitexpr]) {  
    {BLOCK}  
}
```

## Parameters

Parameter	Definition
<i>initexpr</i>	a PSL statement whose evaluation initializes the loop
<i>termexpr</i>	a PSL relational expression whose evaluation determines the continuation or termination of the loop
<i>reinitexpr</i>	a PSL statement whose evaluation reinitializes the loop
BLOCK	one or more PSL statements that are executed once in accordance with the evaluation of the expressions

## Description

The for loop behaves similarly to the C for loop construct in that it is an iteration statement in which the following occurs:

1. The first expression *initexpr* is evaluated once, initializing the loop.
2. The second expression *termexpr* is evaluated *before* each iteration and, if it becomes equal to 0, the for loop terminates.
3. The third expression *reinitexpr* is evaluated *after* each iteration, reinitializing the loop.

The for loop is equivalent to the following example:

```
initexpr;  
while (termexpr) {  
    BLOCK  
    reinitexpr;  
}
```

Generally, *initexpr* and *reinitexpr* are assignments or function calls, and *termexpr* is a relational expression. Any of the three expressions may be omitted; however, the semicolons must remain. A missing second expression *termexpr* makes the implied test equivalent to testing a non-zero constant (or a permanently true expression).

## Example

The following example demonstrates the for loop:

```
for (i = 10; i > 0; i--) {  
    printf(" %d seconds to go\n",i);  
    sleep(1);  
}
```

The output of this example is as follows:

```
10 seconds to go  
9 seconds to go  
8 seconds to go  
7 seconds to go  
6 seconds to go  
5 seconds to go  
4 seconds to go  
3 seconds to go  
2 seconds to go  
1 seconds to go
```

# foreach

Iterates over a list and sets variable to be each element of list, performing BLOCK for each element of list

## Format

```
foreach variable (list) {BLOCK}  
foreach unit variable (list) {BLOCK}
```

## Parameters

Parameter	Definition
unit	controls how <i>list</i> is split into individual elements  <b>Valid Values</b> <ul style="list-style-type: none"><li>• <b>word</b> assumes that the array elements are separated by white space (spaces, tabs, or \n)</li><li>• <b>line</b> assumes that array elements are separated by \n</li></ul> <b>Default</b> <b>line</b>
variable	the name of the element that is equated to each element in <i>list</i>
list	a list that contains one or more elements that can be equated to <i>variable</i>
BLOCK	one or more statements that are executed when <i>variable</i> has been equated to an element from <i>list</i>

## Description

The foreach loop iterates over *list* and sets *variable* to be each element of *list*, performing *BLOCK* for each element of *list* in turn.

## Examples

The following examples highlight the usage of the foreach statement:

### Sum the Elements in an Array

```
sum = 0;
foreach elem ("1\n2\n3\n4\n5") {
    sum += elem;
}
```

### List the Login ID of Each Account on the System

```
foreach user (cat("/etc/passwd")) {
    print (ntharg(item, 1, ":"), "\n");
}
```

---

**Note**

---

cat() and ntharg() are built-in PSL functions.

---

### Count the Number of Words in a String

```
words = 0;
foreach word w ("The cat sat on the mat.") {
    words++;
}
```

# function

Specifies a user-defined function

## Format

```
function name([argument-list]) {BLOCK}
```

## Parameters

Parameter	Definition
name	<p>character label that is used to identify and call the function from within the PSL program</p> <p>The <i>name</i> cannot be the same as a PSL built-in function, or a PSL variable.</p>
argument-list	<p>up to 20 optional PSL variables that are passed to the function as parameters when it is called for execution</p> <p>The <i>argument-list</i> can be a NULL entry if no variables are passed to the function, a single argument, or several arguments separated by commas.</p> <p>Optional ellipses indicating that the function will accept an additional variable number of arguments. This variable portion of the argument list is processed using the <code>va_start()</code> and <code>va_arg()</code> PSL built-in functions. See “Functions with Variable Length Argument Lists” on page 3-15 and the <i>PATROL Script Language Reference Manual Volume 2—PSL Functions</i>.</p>
BLOCK	<p>one or more PSL statements that define the action the function performs</p>

## Description

The function statement provides user-defined functions within PSL programs similar to those available in the C programming language. The function keyword is required in a user function definition.

Two additional keywords, `local` and `return`, are optional:

- `local` declares variables that will be used only within the function.
- `return` identifies function output that is returned to the caller

Functions must be defined before their first use, and the correct *argument-list* must be passed in a function call. A function call always returns a character string representing a character string or numeric value. (All data types are represented within PSL as character strings.)

Arguments are passed-by-value to parameters (that is, local copies are created from the arguments' data passed in), and thus changing a parameter will not affect the value of the argument. Function parameters are local to a function and can have names the same as global variables (or the same as parameters of other functions).

If a function definition appears in the middle of executable statements and control flow reaches that definition from above, the definition is skipped as a comment is skipped. The only way to enter the body of a function is to explicitly call it. The function definitions serve merely to define a function and are not invoked until called. Hence, it is possible to place executable code above, below, and between function definitions.

## return Statement

There are three ways to exit a user-defined function:

- `return` with a return value
- `return` without a return value (return value = NULL string)
- fall through to the bottom right brace (return omitted, no return value)

PSL does not interpret falling through the bottom of a function as an error condition, although BMC Software does not recommend relying on “fall-through” for a function whose return value is used.

PSL produces a compilation warning similar to that produced by C compilers when it encounters return statements within a function of which some have return values and while others do not. Having multiple exit points in a function that exit in different ways may indicate confusion over whether the function was defined to perform an action or return a value.

BMC Software recommends that you design functions that return a value to explicitly return the NULL string when they have no other value to return. This is preferred to exiting the function with a `return` statement and no return value.

## Functions with Variable Length Argument Lists

The use of the ellipsis (...) in a `function()` statement indicates that the function will accept zero or more additional arguments. A `function()` statement that contains ellipses is said to be a *variable length argument list* function.

Following is an example of a variable argument list function that also has two fixed arguments, `argno1` and `argno2`:

```
function myfunc(argno1, argno2, ...) {BLOCK}
```

Because `argno1` and `argno2` are fixed arguments, they must be included in every call to `my_func()`. The ellipsis, however, indicates that any number of additional arguments may also be included in a call to `my_func()`. For example, the following are legal calls to `my_func()` because each includes two or more arguments:

```
my_func(1, 2);  
my_func(1, 2, 3, 4, 5);
```

The following, however, is not a valid call to `my_func()` because it does not include the minimum two fixed arguments:

```
my_func(1); # illegal function call!
```

You can process a variable argument list within your function() statement using the PSL `va_start()` and `va_arg()` functions:

- The `va_start()` function initializes the variable argument list to return the first argument. You can use the `va_start()` function multiple times within a function() statement to initialize the variable argument list, allowing the function() statement to traverse the variable argument list more than one time.
- The `va_arg()` function returns the current argument in the variable argument list and increments. Successive `va_arg()` function calls return successive arguments from the variable argument list until all arguments have been returned. Unless you call the `va_start()` function to reinitialize the variable argument list, the `va_arg()` function returns a NULL after all variable arguments have been returned.

See the *PATROL Script Language Reference Manual Volume 2—PSL Functions* for descriptions and examples of the `va_arg()` and `va_start()` built-in functions.

## Defining Local Variables

User-defined function local variables are declared using the `local` keyword inside the body of the function. The `local` keyword declares one or more variables specified in a comma-separated list that is terminated by a semicolon. These names become local variables to the function. Following is an example of local variable definitions:

```
function f() {  
    local x;  
    local a,b;  
    # . . . Statements for the function execution.  
}
```

Local variables cannot have the same name as a function parameter or another local variable in the same function. Local variable names in one function do not affect those in another function. Local variables can have the same name as a global variable and can “hide” a global name this way. BMC Software does not recommend that you use local variables this way, and PSL will generate a compiler warning each time it detects this situation.

Local variable declarations are treated as expressions and can appear anywhere within the function that an expression is valid. However, there is no concept of inner scopes in inner blocks, and a local variable has scope extending from its point of declaration to the end of the enclosing function (not the enclosing block). BMC Software recommends that you declare all local variables at the start of the function body.

Local variables are initialized to the empty string every time the function is entered. They do not retain their values from a previous call.

Each user-defined function (except for the `main()` function) permits a maximum of 20 local variables.

## Entry Point Function

In earlier releases of PSL, the program entry point was always the first PSL statement. Although the current PSL still supports this concept, the addition of user-defined functions defined at the beginning of a program causes a need for other possible entry points. One of these is the PSL entry point function.

The PSL entry point function is equivalent to the C language `main()` function. If a PSL program contains a user-defined function named `main`, execution begins at the first statement in `main()`. The PSL program terminates normally when `main()` returns.

You can specify that a user-defined function with a label other than `main()` be treated as the main program entry point by specifying the `-e` option to the PSL stand-alone compiler. Refer to Chapter 4, “PSL External Commands,” for a description of the PSL compiler command and options. The function you specify as the entry point is permitted to have the same properties as `main()`.

The `main()` function or the entry point function must be defined in the top-level PSL program and not in any imported libraries. Functions imported from libraries are ignored when determining whether an entry point function is available.

## Start of Execution Without an Entry Point Function

If there is no `main()` function and no entry point function specified using the PSL compiler `-e` option, execution begins at the first executable statement that is not inside a function definition. This behavior maintains backward compatibility with previous versions of PSL.

A program without an entry function will normally have function definitions at the top (they must be defined before their first use) and the main executable statements afterwards. A typical example would be the following:

```
function max(x,y) {
    if(x > y) {
        return x;
    } else {
        return y;
    }
}
m = max(1,2); # Execution starts here
print("maximum is ", m, "\n");
```

As indicated, program execution begins immediately after the function definition.

## Backward Compatibility with Earlier PSL Versions

Almost all programs written in earlier versions of PSL should function identically under the new PSL version. Notably, there is no need to change older PSL scripts to have a `main()` function or entry point function since the current PSL version supports and defaults to older style of execution at the first statement.

Run-time errors may occur if you attempt to migrate a program written using the current version of PSL back to an earlier version. Older versions of PSL will not recognize the following function keywords:

- `function`
- `local`
- `return`

## Limitations of User-Defined Functions

User-defined functions are subject to the following limitations.

### Function Calls Are Non-Recursive

User-defined functions can make unlimited calls to other functions provided that there is no direct or indirect recursion in the sequence of calls.

PSL user-defined functions do not support recursion because each function has only one block of memory for its parameters and local variables. A recursive call would overwrite the parameters for the previous (and still active) call.

PSL prevents direct recursion by generating a compilation error. The PSL compiler may compile and execute a program with indirect recursion, but the potential for overwriting the parameters and variables for an active function will produce unexpected results.

## **Argument Pass by Reference Not Supported**

PSL functions do not support argument passing by reference; only argument pass-by value is supported. PSL also has no concept of pointers, which are used in the C programming language to mimic pass-by reference.

## **Parameter and Local Variable Limits**

PSL functions, including `main()`, have the following parameter and local variable limits:

- maximum of 10 parameters
- maximum of 20 local variables

## **Function Nesting Not Permitted**

PSL does not permit function nesting—each function definition must be at global scope and cannot be defined inside any other function.

# if

Conditionally executes a **BLOCK** of PSL code

## Format

```
if (expression)
    {BLOCK}

if (expression)
    {BLOCK} else {BLOCK}

if (expression)
    {BLOCK}
elsif (expression)
    {BLOCK}
. . .
else
    {BLOCK}
```

## Parameters

Parameter	Definition
expression	a PSL statement whose evaluation returns either TRUE or FALSE
BLOCK	one or more PSL statements that are executed once in accordance with the evaluation of the <i>if</i> or <i>elsif</i> expressions

## Description

The if statement conditionally executes a **BLOCK** of PSL code. The if statement is straightforward. Since a statement **BLOCK** is always bounded by braces, there is no ambiguity about which if, elsif, and else goes with.

# Examples

The following examples highlight the usage of if, elsif, and else:

## if statement

```
if (x > 10) {  
    x = 10; # don't let x get bigger than 10  
}
```

## if . . . else Statement

```
if (x == 0) {  
    # do something  
} else {  
    # x != 0  
    # do something else  
}
```

## if . . . elsif . . . else Statement

```
if (x == 0) {  
    # do something  
} elsif (x == 1) {  
    # do something else  
} else {  
    # x != 0 && x != 1  
    # do something else  
}
```

# last

Causes a PSL process to exit the innermost execution loop

## Format

```
last;
```

## Parameters

This statement has no parameters.

## Description

The `last` statement causes PSL execution to exit the innermost execution loop. The `last` statement is equivalent to the C `break` statement. The `last` statement must be terminated with a semicolon when used in a PSL program.

# next

Immediately starts the next iteration of the innermost execution loop

## Format

```
next ;
```

## Parameters

This statement has no parameters.

## Description

The `next` statement immediately starts the next iteration of the innermost execution loop. The `next` statement is equivalent to the `C continue` statement.

# requires

Imports variables and functions from a PSL library

## Format

```
requires library;
```

## Parameter

Parameter	Definition
library	name of the library whose specified <i>export</i> variables and functions are to be imported into the PSL program

## Description

The `requires` statement imports variables and functions identified in `export` statements from a previously created PSL library into the PSL program. Each `requires` statement can specify a single library name.

PSL contains no explicit import statement; using the `requires` statement implies importation. The `requires` statement searches for the binary containing the library and reads all its `export` statement information, then imports the specified variables and/or functions into the PSL program.

Any number of `requires` statements can appear in a PSL program. All libraries specified in `requires` statements must be available to the compiler during compilation.

## requires Statements in Imported Libraries

The PSL compiler will automatically resolve nested dependencies in imported libraries, but it will not automatically load all the other exported functions and variables found in the library that satisfies the nested dependency. You must explicitly import a library in order to guarantee access to all the exported variables and functions within it.

A `requires` statement can appear inside a function definition without special significance. BMC Software discourages placing `export` statements inside function definitions and recommends that you place all `requires` statements at the top of the file.

---

### Note

---

BMC Software recommends that you minimize the use of `export` statements in libraries where the exported variables and functions depend on variables and functions imported with `requires` statements from other libraries. Following this practice can avoid chains of import dependencies and the possibility of libraries that form `requires` statement loops.

---

## Variable and Function Availability Among Imported Libraries

When a PSL program imports variables and functions from more than one library, the imported variables and functions from one library can set and use the imported variables and functions from the others, regardless of how the libraries are loaded for compilation.

BMC Software strongly recommends that you avoid the practice of creating and importing mutually referential libraries. Mutually referential libraries are those that contain a `requires` statement naming the other library.

## Errors Involving the `requires` Statement

The `requires` statement can generate compiler errors in the following instances:

- A reference to an imported variable or function appears before the requires statement that imports it. You must place a requires statement before the first use of the imported variable or function.
- An imported function has the same name as a function defined within the PSL program.
- The same variable or function name is imported from two or more libraries.

# switch

Executes a specific BLOCK of PSL code based on the value of a variable

## Format

```
switch (expression)
{
  case a: {BLOCK}
  case b: {BLOCK}
  . . .
  case p,q,r: {BLOCK}
  . . .
  case n: {BLOCK}
  default: {BLOCK}
}
```

## Parameters

Parameter	Definition
<i>expression</i>	a PSL expression whose integer value specifies the PSL statement <i>BLOCK</i> that will be executed
<i>a,b, ... p,q,r, ... n</i>	an integer values indicating the value of <i>variable</i> that will cause the corresponding <i>BLOCK</i> to be executed
<i>BLOCK</i>	one or more statements that are executed when the corresponding <i>case</i> value equals <i>variable</i>

## Description

The switch statement evaluates *expression* and based on its integer value executes a specific PSL *BLOCK*. The case labels correspond to the values of *expression* for which a specific PSL *BLOCK* is available.

If the value of *expression* falls outside the range of the values in the case labels, execution continues with the *BLOCK* corresponding to the default label. If no default label exists, execution will continue with the first statement following the switch statement.

The switch statement is similar in form and function to the C `switch` statement.

The PSL `switch` statement executes in almost the same way as a long sequence of if-then-else-if statements. A case or default clause is effectively a run-time statement that specifies a comparison against the value of *expression*:

- If the value of *expression* matches a case, execution moves inside the *BLOCK* for the case or default clause; and after completing *BLOCK*, execution continues after the entire switch statement (that is, there is no falling through to the next case clause).
- If the value of *expression* does not match a case, execution skips to the default clause; and if there is none, execution moves to the statement following the `switch` statement.

Any statement within the `switch` statement case block that is not part of a case or default *BLOCK* executes only if all the case labels above it failed to match *expression* (that is, it executes as part of the normal sequence of control flow).

## Differences between the PSL and C `switch` Statements

The following differences exist between the PSL `switch` statement and the C `switch` statement:

- PSL case expressions can be dynamically evaluated expressions whereas C only permits constant expressions.
- The colon delimiter that separates the case label from the executable *BLOCK* is optional in PSL and required in C.

- PSL requires that the default label follow all case labels in the switch statement case block, whereas C allows `default` to appear anywhere within the case labels. PSL returns a compilation error if one or more case labels follow `default`.
- PSL does not return a compilation error for duplicate case labels in the switch statement case block, whereas C does. In PSL, the second of the duplicate case labels is unreachable.
- PSL allows multiple cases that execute a common *BLOCK* to be specified as a comma separated list within a single case label, where C requires that each case be a separate `case` label stacked above a single *BLOCK*. (Conversely, the stacked labels will not work in PSL.)
- Execution of a PSL *BLOCK* does not “fall through” to the next case label and *BLOCK*, as it does the C `switch` statement. Upon reaching the closing right brace of a case or default *BLOCK*, execution moves to the end of the PSL switch statement.
- The PSL switch statement uses the last statement to exit from a *BLOCK*, whereas C uses the `break` statement. The last statement exits the innermost switch statement or loop. However, because of the absence of “fall-through” in PSL, there is little need to use the last statement in the switch statement.

## Similarities between the PSL and C switch Statements

The following similarities exist between the PSL switch statement and the C `switch` statement:

- both generate a compiler error upon detecting two default labels in a single switch statement
- both permit nested `switch` statements

## Efficiency of switch Statements versus If-Then-Elsif Sequences

Because of the similar method of implementation, there is almost no difference in efficiency between PSL `switch` statements and `if-then-elsif` sequences. Programming style is the main consideration in the choice. To speed up `switch` statements, BMC Software recommends that you specify the most likely cases first. The speedup is also true of `if-then-elsif` sequences.

### Pitfall: switch Statement case Labels That Modify case Variables

The `case BLOCKs` are evaluated at run-time in their order of appearance:

- case order for `BLOCKs`
- left-to-right for expressions in the comma-separated lists of multiple-case labels

All expressions within a comma-separated list are evaluated before the case label. This evaluation occurs even if the first expression is a match.

This sequence and method of evaluating the case label can be a dangerous pitfall if any expression in the list modifies either *variable* for the current `switch` statement or a variable used in another `case` expression.

### Pitfall: Statements Inside a switch Statement That Are Not Part of a BLOCK

Under PSL, statements within a `switch` statement that are not part of a `BLOCK` (*free* statements) can and will be executed if they are reached by the flow of execution. The condition for control flow to reach these statements is that *variable* cannot match any of the `case` labels that precede them within the `switch` statement.

### Pitfall: Nesting case Labels That Use the Same Variable

PSL does not return a warning or error message when two case labels evaluated against *expression* are nested one inside the other. Two examples of this situation are shown in the following PSL `switch` example:

```
switch(x)
```

```

{
  case 1:
  {
    f1()                # Function f1 Called
    case 2 : {f2();}    # Function f2 Unreachable
    f3();                # Function f3 Called
  }
  default: {case 4: {f4();}} # Function f4 called if x=4
}

```

Since case and default labels are run-time statements, the effect of one case label nested within another is that *expression* must match the case value for the case *BLOCK* to execute. This means that *expression* must equal two different values! In case 1 of the example, *f2* will never be called because *x* cannot equal both 1 and 2.

In the default case of the example, *f4* will be called if *expression* = 4 because there is no case 4 defined in the switch statement. When *expression* = 4, the default *BLOCK* executes, containing the case 4 *BLOCK* call to function *f4*.

Although nesting case labels within one another is possible and may have some utility, BMC Software views them as a potential pitfall both because of the possibility of creating unreachable *BLOCKS* and because future PSL versions may not support case label nesting.

# while

Executes a BLOCK of PSL code while the evaluation of a statement is TRUE

## Format

```
while (expression) {BLOCK}
```

## Parameters

Parameter	Definition
expression	a PSL statement whose evaluation returns either TRUE or FALSE
BLOCK	one or more PSL statements that execute repeatedly as long as <i>expression</i> evaluates to TRUE

## Description

The while loop executes *statements* as long as *expression* evaluates to TRUE (non-zero).

## Example

The following sample PSL statements print the integers from 1 to 10:

```
x = 1;
while (x <= 10) {
    print (x, " ");
    x++;
}
print ("\n");
```



---

# PSL External Commands

PSL provides several commands external to the language that allow you to execute PSL commands from the operating system command line and to compile and execute PSL programs. The following topics are discussed:

%DUMP—List Specific Information . . . . .	4-2
%DUMP CHANNELS—List PSL Global Channels . . . . .	4-3
%DUMP LIBRARIES—List Loaded PSL Libraries . . . . .	4-4
%PSL—Execute a PSL Statement . . . . .	4-6
%PSLPS—List Current PSL Processes . . . . .	4-7
psl—PSL Compiler Command . . . . .	4-8

# %DUMP—List Specific Information

Return a list of specific information

## Format

`%DUMP option`

## Parameters

This command has no parameters.

## Description

The %DUMP command returns a list of information that is specified by the parameter option.

Field	Definition
option	<ul style="list-style-type: none"><li>• ALL—returns info about PATROL Agent data structures</li><li>• APP_INSTS—returns info about each application instance</li><li>• APPS—returns a list of applications</li><li>• CHANNELS—returns a list of open PSL global file and process channels</li><li>• CONSOLES—returns a list of connected consoles</li><li>• ERRORS—returns a list of PSL errors that have occurred</li><li>• GLOBALS—returns a list of global channels</li><li>• KM_LIST—returns a list of loaded KMs</li><li>• LIBRARIES—returns a list of loaded libraries</li><li>• PARAMS—returns a list of PATROL Agent parameters</li><li>• RTLIST—returns info about processes in the Agent run-time queue</li><li>• RUNQ—returns a list of items scheduled in the run queue</li><li>• TASKS—returns a list of current tasks</li></ul>

# %DUMP CHANNELS—List PSL Global Channels

List open PSL global process and file channels

## Format

```
%DUMP CHANNELS
```

## Parameters

This command has no parameters.

## Description

The %DUMP CHANNELS command returns a list of global file and process channels opened using either the `fopen()` or `popen()` functions. The output from the %DUMP CHANNELS command contains the same information as that provided by the `get_chan_info()` function within a PSL program. The PSL interpreter returns the list of global channel information to the console window from which the %DUMP CHANNELS command was executed.

---

### Note

---

The %DUMP CHANNELS command is one of a series of dump commands available from the command line. Use the %DUMP command (no argument) to obtain a list of items that can be dumped.

---

Each line of output from the %DUMP CHANNELS command is a string with the format:

```
name status details type readname readpid writepid writename
```

Field	Definition
name	alphanumeric name given to the channel when it was created as a global channel, or when it was changed from local channel to a global channel

Field	Definition
status	OPEN or CLOSED
details	one of the following <ul style="list-style-type: none"> <li>• <code>fopen()</code> channel—file name that is opened or NONE if no file name is open</li> <li>• <code>popen()</code> channel—process ID of the external operating system process to which the channel is attached or -1 if the process has terminated</li> </ul>
type	PIPE or FILE
readname	one of the following <ul style="list-style-type: none"> <li>• name of the process waiting to read from the channel</li> <li>• NONE if no process is waiting</li> <li>• UNAVAILABLE if there is a process but the name is not available</li> </ul>
readpid	one of the following <ul style="list-style-type: none"> <li>• process ID of the PSL process waiting to read from the channel</li> <li>• -1 if no process is waiting</li> </ul>
writepid	one of the following <ul style="list-style-type: none"> <li>• process ID of the PSL process waiting to write to the channel</li> <li>• -1 if no process is waiting</li> </ul>
writename	one of the following <ul style="list-style-type: none"> <li>• name of the process waiting to write to the channel</li> <li>• NONE if no process is waiting</li> <li>• UNAVAILABLE if there is a process but the name is not available</li> </ul>

## %DUMP LIBRARIES—List Loaded PSL Libraries

Display a list of libraries that the Agent has loaded

### Format

```
%DUMP LIBRARIES
```

## Parameters

This command has no parameters.

## Description

Each line of output from the %DUMP LIBRARIES command is a string with the format:

*library\_name - status, date*

Field	Definition
library_name	the name of the loaded library
status	indicates if the library has been modified on disk since it was loaded by the agent
date	the last modification date the library had when it was loaded by the agent

## Example

When typed into the computer window, the command  
OS>%DUMP LIBRARIES outputs the following:

```
===== Currently Loaded Libraries =====  
response_def_lib.lib - unmodified, Wed Nov 6 15:34:42 1996  
unix_misc_lib.lib - unmodified, Wed Dec 11 13:11:16 1996  
set_share_lib.lib - unmodified, Tue Nov 26 16:56:38 1996  
=====
```

# %PSL—Execute a PSL Statement

Execute a one-line PSL statement

## Format

`%PSL statement`

## Parameter

Parameter	Definition
statement	one or more PSL statements or built-in functions that are to be executed  The %PSL command and <i>statement</i> must fit on a single input line.

## Description

The %PSL command submits *statement* to the PSL interpreter for immediate interpretation.

# **%PSLPS—List Current PSL Processes**

List all scheduled or active PSL processes on the computer system

## **Format**

`%PSLPS`

## **Parameters**

This command has no parameters.

## **Description**

The `%PSLPS` command returns a list of currently scheduled or active PSL processes on the computer system to the console window. The information displayed by the `%PSLPS` command includes the process identifier (PID) and the name of the PSL process or command.

# psl—PSL Compiler Command

Call the PSL interpreter and compiler for the specified PSL source file

## Format

```
psl
psl inputfile [-o outfile -n -r -w -l -P -O -q -b -R
               -h -v -e functionname -s librarynames -S]
```

# Parameters

Parameter	Definition
	<p>no parameters</p> <p>Entering this command without parameters starts an interactive mode of the PSL compiler. It executes everything that you type at a prompt that precedes the end of file (EOF) character, which is <b>Ctrl+d</b> or <b>Ctrl+c</b> on most platforms.</p>
inputfile	name of the PSL source file
-o outfile	<p>write the compiled binary output as the file <i>outfile</i></p> <p>This option implicitly includes the <code>-n</code> option; that is, the <code>-o</code> option does not schedule <i>outfile</i> for execution after compilation.</p> <p>The PSL compiler may add a <code>.lib</code> or <code>.bin</code> extension to outfile if you selected the <code>-l</code> or <code>-b</code> option respectively.</p>
-n	do not schedule <i>outfile</i> for execution after compilation
-r	<p>suppress run-time error messages</p> <p>This option is equivalent to including the statement <code>Ps1Debug = 0;</code> in the PSL script.</p>
-w	suppress compilation warning messages produced by the PSL compiler
-l	<p>write the compiled output in library mode as the file <i>outfile.lib</i>.</p> <p>This option implicitly includes the <code>-n</code> option; that is, the <code>-l</code> option does not schedule <i>outfile.lib</i> for execution after compilation.</p>
-P	activate the PSL Profiler
-O	<p>specify the optimizer level</p> <p>For more information about optimizer levels, see “Optimization Levels” on page C-11.)</p>
-q	print the PSL bytecode to the screen

Parameter	Definition
-b	<p>write the compiled output in binary mode as the file <i>outfile.bin</i></p> <p>This option implicitly includes the <code>-n</code> option; that is, the <code>-b</code> option does not execute the binary after compilation.</p>
-R	<p>inform the PSL compiler that statically or dynamically loaded libraries are required for the compilation</p> <p><b>Default</b> No statically or dynamically loaded libraries are required for the compilation.</p>
-h	display command line help information
-v	display version number information
-e <i>functionname</i>	<p>specify the user-defined function <i>functionname</i> within the PSL source file as the execution entry point for the compiled program</p> <p><b>Default</b> Execution begins with the first statement within <i>inputfile</i> that is not part of a user-defined function definition.</p>
-s	<p>instruct the PSL compiler to statically load all user libraries required for compilation</p> <p><b>Default</b> Load library names with the <code>.lib</code> extension statically and all other library names dynamically.</p>
<i>librarynames</i>	<p>list of user library names that should be statically loaded for compilation</p> <p>Library names should have the format <i>library_name.lib</i>. The PSL compiler loads any library name without the <code>.lib</code> extension dynamically unless the <code>-l</code> flag is specified.</p>
-S	display the PSL symbol table

## Description

The `psl` command creates an executable binary file from *inputfile* and specified *librarynames*. The stand-alone interpreter will immediately execute the compiled binary if options `-o outfile`, `-l`, and `-b` are omitted. Use this command to create PSL libraries and PSL binaries that can be used in the PATROL Agent. Doing so allows for a significant performance gain.

The following types of PSL functions are not supported by the stand-alone PSL interpreter:

- functions that rely on the object hierarchy of the PATROL Agent such as `create()`, `destroy()`, `history()`, `get_ranges()`, and other functions
- functions that rely on multiple PSL processes such as the `lock()` function
- SNMP functions
- event management functions
- `response()` functions

Other PSL functions that do not rely on the PATROL Agent such as file manipulation, string manipulation, and print functions will work.



---

# Diagnosing PSL Program Errors

This chapter provides information on diagnosing PSL errors. The following topics are discussed:

PslDebug—Run-Time Error Checking Variable . . . . .	5-2
errno—Error Return Code Variable . . . . .	5-6
exit_status—System Return Code Variable . . . . .	5-7
Incompatibilities with the C Programming Language . . . . .	5-7
Operators && and    . . . . .	5-7
Prefix and Postfix Operators ++ and -- . . . . .	5-8
Break and Continue Statements . . . . .	5-8
Common PSL Coding Errors . . . . .	5-8
Character Strings Interpreted as Numbers . . . . .	5-9
Floating Point Numbers Interpreted as Character Strings . . . . .	5-10
Character Strings Interpreted as Variable Names . . . . .	5-10
PSL Functions That Do Not Modify Their Arguments . . . . .	5-11
Functions That Do Not Write to the Console Window . . . . .	5-12
PSL Compiler Warnings . . . . .	5-12
Built-in Function Run-Time Error Messages . . . . .	5-15

# Ps1Debug—Run-Time Error Checking Variable

The `Ps1Debug` built-in local variable provides a run-time error checking and trace facility that detects a variety of common PSL coding errors. You can enable or disable run-time checking and/or tracing by equating `Ps1Debug` to a numeric value representing 1 or more of 16 error-checking flag bits.

The trace data produced by the `Ps1Debug` trace flags is useful in debugging PSL programs during the initial stages of programming or as a “pre-debug” before calling the more powerful (and hence more time consuming) PSL debugger.

The general arrangement of the flags is as follows:

- error checking: 0-128
- tracing: 256-16384

The format of `Ps1Debug` within a PSL script is identical to that of any other PSL variable:

```
Ps1Debug = n;
```

where *n* is the sum of one or more of the values from Table 5-1 on page 5-3.

**Table 5-1 PslDebug Error Checking Flag Bits (Part 1 of 3)**

Value	Definition
<b>PslDebug Enable All Error Checking and Tracing</b>	
<b>-1</b>	enable all PSL run-time error checking and tracing functions  Setting <code>PslDebug = -1</code> is equivalent to setting all error checking and tracing flag bits, or setting <code>PslDebug = 257</code> ( <code>1 + 256</code> enables all error checking and all tracing flag bits).
<b>PslDebug Enable/Disable All Error Checking</b>	
<b>0</b>	disable all run-time error checking
<b>1</b>	enable all PSL run-time error checking  Setting <code>PslDebug = 1</code> is equivalent to setting all error checking flag bits.

**Table 5-1 PsIDebug Error Checking Flag Bits (Part 2 of 3)**

Value	Definition
<b>Numeric Operation Warnings</b>	
2	<p>enable warnings when arithmetic operations involve NULL string operands                      This check is a stylistic one used to verify that numeric variables are explicitly initialized to zero rather than defaulting to NULL strings, which are treated as zero in arithmetic operations.</p> <p><b>Example</b>                      The expression <code>x+10</code> will generate this warning when <code>x</code> is the NULL string.</p>
4	<p>enable warnings for arithmetic operations involving nonnumeric operands</p> <p><b>Example</b>                      The expression <code>"mary" + "john"</code> attempting to numerically add two character strings will generate this warning.</p>
8	<p>enable warnings for illegal or undefined arithmetic operations</p> <p><b>Example</b>                      Divide by zero.</p>
<b>Variable Initialization Warnings</b>	
16	<p>enable warnings for variables that were not explicitly initialized                      This checks for variables that are used before being explicitly equated to a value. Variables are initialized to the NULL string by default. BMC Software recommends that you explicitly initialize all variables to a value (even the NULL string) as a matter of good programming style.</p>
<b>Built-in Function Warnings</b>	
32	<p>enable warnings for PSL Version 2.0 built-in functions                      This check allows you to suppress warning messages that were not suppressible in PSL Version 2.0.</p> <p><b>Example</b>                      A "file not found" for the <code>cat()</code> function will generate this warning.  <code>PsIDebug = 32</code> is the default if <code>PsIDebug</code> is not specified to provide the same default behavior as PSL Version 2.0.</p>
64	<p>enable warnings for PSL Version 3.0 built-in functions                      This is a new error check of built-in functions that were not in PATROL Version 2.0. The errors detected include passing non-numeric arguments when a numeric value is required, passing a bad argument, or passing the wrong number of arguments to a <code>printf()</code> function.</p>
128	not used
<b>PsIDebug Enable All Tracing</b>	
256	<p>enable all PSL run-time tracing. Setting <code>PsIDebug = 256</code> is equivalent to setting all tracing flag bits</p>

**Table 5-1 PslDebug Error Checking Flag Bits (Part 3 of 3)**

Value	Definition
<b>Function Tracing</b>	
<b>512</b>	enable function call tracing Function call tracing reports which functions are called but does not return information about the arguments. Function call tracing traces both user-defined and built-in functions.
<b>1024</b>	enable function argument tracing Function argument tracing reports the arguments passed to all user-defined or built-in functions. Function argument tracing requires that function call tracing ( <code>PslDebug = 512</code> ) also be enabled; that is, <code>PslDebug = 1536</code> for function argument tracing.
<b>2048</b>	enable function return value tracing Function return value tracing reports the value returned by calls to all user-defined or built-in functions.
<b>Variable Tracing</b>	
<b>4096</b>	enable variable assignment tracing Assignment tracing reports the variable name (if available) and the value assigned to it.
<b>8192</b>	enable errno tracing The errno tracing reports any nonzero values stored in the PSL <code>errno</code> variable.
<b>Lock Tracing</b>	
<b>16384</b>	enable PSL lock tracing Lock tracing reports the interprocess actions that occur during <code>lock()</code> and <code>unlock()</code> function processing, including the granting, denying, and releasing of locks.
<b>32768</b>	not used

You set multiple flag bits by equating `PslDebug` to the sum of their values. For example, `PslDebug = 44` would enable the following flags:

- arithmetic operations involving nonnumeric operands (4)
- illegal or undefined arithmetic operations (8)
- PSL Version 2.0 built-in functions (32)

whose sum  $4 + 8 + 32$  is 44.

# errno—Error Return Code Variable

The PSL `errno` variable is set by various PSL built-in functions to indicate the reason for a failure. All functions that can potentially set `errno` are also required to reset the `errno` variable to zero and have it remain zero if the function is successful. Functions that don't set the `errno` variable do not reset it to zero either. This requirement validates usage styles such as the following:

```
x = cat("file");
if (errno != 0)
{
    # Error occurred in cat function
    # examine errno value for specific error code
}
```

The user can also write values to the `errno` variable. However, there should be little cause to set `errno` since it is reset by all functions that set it.

---

### Tip

---

The `errno` variable is reset to zero by many built-in functions at the start of their processing; and in some functions, clearing can occur before arguments are processed, leading to `errno` being cleared before being processed as an argument. Hence, when passing `errno` as an argument to a built-in function, BMC Software recommends that you make a copy in a temporary variable and pass the copy.

---

# exit\_status—System Return Code Variable

The PSL `exit_status` variable stores the exit status of a process invoked from a `system()`, `popen("OS",)` or `execute("OS",)` function. The user cannot write the `exit_status` variable.

---

## Tip

---

The `popen()` function with an OS command type can set the `exit_status` variable asynchronously whenever its operating system child process dies. It is possible (although unlikely) that the `popen()` function could set `exit_status` after a `system()` or `execute()` function concludes but before `exit_status` is read for the `system()` or `execute()` call.

---

## Incompatibilities with the C Programming Language

There are features found in the C programming language that are not supported by and not compatible with PSL. This section lists known incompatibilities.

### Operators `&&` and `||`

**Difference:** PSL does not perform short-circuit processing of the `&&` and `||` operators.

**C Action:** The `&&` and `||` functions short-circuit in standard C meaning that evaluation stops with the first condition that establishes an outcome for the statement. Some programmers use this feature to construct statements that depend on the left-to-right sequential evaluation of the conditions.

**PSL Action:** PSL always evaluates all conditions in a statement using `&&` and `||` operators. Evaluation of all conditions may cause unexpected results in statements that depend on a left-to-right sequential evaluation and short-circuit.

## Prefix and Postfix Operators ++ and --

Difference: PSL does not distinguish between prefix and postfix operators.

C Action: Appending ++ or—as a prefix instructs the program to increment or decrement the variable then test the condition, while appending ++ or—as a postfix instructs the program to test the condition before incrementing or decrementing the variable.

PSL Action: PSL treats both prefix and postfix operators as prefix operators. Treating postfix operators as if they were prefix operators may cause unexpected results in statements that depend on postfix operators.

## Break and Continue Statements

Difference: PSL supports the function but not the syntax of the `break` and `continue` statements. PSL replaces the `break` statement with the last statement, and the `continue` statement with the next statement.

PSL Action: PSL treats the C `break` and `continue` statements as statements that evaluate the variable name *break* or *continue* and produce no output or effect within the PSL program.

## Common PSL Coding Errors

This section describes common PSL coding errors that result in PSL compiler, interpreter, or run-time errors.

## Character Strings Interpreted as Numbers

**Problem:** Errors can occur when numeric operations are applied to character strings whose first character is a digit or minus sign because PSL will evaluate these character strings as numbers.

**Reason:** Before PSL performs an operation, it must determine whether the operands are numeric or character. To make the determination, PSL performs one of the following tests:

- For relational operators, PSL tests that both operands consist entirely of digits, a period, and/or a minus sign. Otherwise, both operands are character type.
- For arithmetic operators, PSL tests only that the first character of each operand is a digit or minus sign. Otherwise, both operands are character type that generates a run-time warning.

The run-time warnings perform the same check as the arithmetic operators.

---

### Note

---

These features remain for compatibility with previous releases of PATROL.

---

Arithmetic operators identify the character string “10th May” as the number 10 while relational operators correctly identify it as a character string. Its use as an arithmetic operand will not generate a run-time warning.

**Solution:** Be aware that character strings whose first character is a valid numeric representation can cause unexpected results in numeric operations that will not be detected by PSL diagnostics. Avoid the use of such strings.

## Floating Point Numbers Interpreted as Character Strings

**Problem:** PSL interprets floating point numbers with a leading decimal point as character strings.

**Reason:** Before performing a numeric operation, PSL evaluates both operands to verify that they are numbers. PSL tests only that the first character of each operand is a digit or minus sign. Upon detecting that the first character is a decimal point, PSL identifies the operator as a character string and returns a warning.

PSL identifies the operand 0.33 as a number and the operand .33 as a character string.

**Solution:** Begin each floating point number with a digit or a minus sign.

## Character Strings Interpreted as Variable Names

**Problem:** PSL interprets character strings that are not enclosed in double quotation marks as variable names.

**Reason:** The PSL language requires that all character strings appearing in statements and function calls be enclosed in double quotation marks to identify them as character strings.

Improperly identified character strings will not generate compiler errors but may generate warnings about uninitialized variables if the string is not equivalent to a variable name initialized elsewhere in the program.

PSL interprets the statements `get(Dev)`; and `get(RDB/Dev)`; as requests for the values of the *user* variables `Dev` and `RDB/Dev`. PSL interprets the statements `get("Dev")` and `get("RDB/Dev")` as requests for the *PSL object* variables `Dev` and `RDB/Dev`.

**Solution:** Enclose all strings in double quotation marks. Enable `PslDebug` flag 16 to catch variable names that were not explicitly initialized within the PSL program.

## PSL Functions That Do Not Modify Their Arguments

**Problem:** Some PSL functions do not modify their arguments, and their return values are lost if not explicitly saved in the PSL program.

**Reason:** Some PSL functions return modified copies of their arguments as return values, preserving the integrity of the original arguments.

For example, the PSL function call `trim(text, "\t");` returns a copy of the character string variable `text` with all tab characters removed. Without an explicit destination and because the `trim` function cannot modify its `text` argument, the return value is discarded before executing the next PSL statement. Alternately, the PSL function call `text=trim(text, "\t");` returns the modified copy of string variable `text` and stores it back into `text`.

PSL functions that do modify their arguments are those that perform command execution and PSL object manipulation, including the following:

- `create()`
- `destroy()`
- `system()`
- `execute()`
- `set()`
- `log()`
- `close()`
- `print()`
- `popen()`
- `write()`

The return values for these functions generally indicate the completion status of the operation and may not need to be retained.

**Solution:** Become familiar with the descriptions of the PSL built-in functions. Those that return more than a completion status value probably do not modify their arguments and thus require that you explicitly save the return value if it is to be available to other statements within the program.

## Functions That Do Not Write to the Console Window

**Problem:** PSL built-in functions such as `grep()`, `cat()`, or `get_var()` do not display their return values in the system console window.

**Reason:** PSL built-in functions return their values to the PSL program. To display those values outside the program, they must be printed.

For example, `grep()`, `cat()`, and `get_var()` produce no output to the system console window but `print(grep())`, `print(cat())`, and `print(get_var())` will print the specified function return values.

**Solution:** To display function return values in the system console window, make those functions arguments of the `print()` function.

## PSL Compiler Warnings

The PSL compiler offers a number of compiler checks for common PSL coding errors. These warnings can be ignored because they do not prevent successful compilation and execution of the PSL program. Currently, these warnings are only available through the **Check Syntax** menu entry in the PSL editor on the PATROL Console.

### *line\_number* **Assignment in if statement**

**Problem:** The assignment operator “=” appeared in an **if** statement condition.

**Reason:** The usual assignment operator within an **if** statement condition is the test equal operator “==”. Using “=” instead of “==” may be a useful coding technique or may indicate an error.

**Solution:** Verify that the **if** statement condition operator is the desired one.

*line\_number* **Statement has no effect**

**Problem:** The PSL statement is a NULL effect statement without assignment, increment, or side-effect function calls.

**Reason:** The statement may produce an effect during execution but does not transmit that effect to any other part of the PSL program, which is the reason the statement is considered a NULL effect statement. An example with operators is `x+2;` which should probably be `x+=2;`. Another example of errors with function calls is `trim();` which should probably be `var=trim();` because the returned value is unused.

**Solution:** Examine the statement and modify it to give it an effect within the program.

*line\_number* **Variable used but not set**

**Problem:** A local PSL variable has been used without ever having been initialized to a value.

**Reason:** Flags within the `Ps1Debug` variable allow PSL to flag all local variables whose first appearance in a program is not as the recipient of an assignment statement. When a variable is created, its value defaults to the NULL string. This message is warning you that the variable in question has not been previously defined within the PSL program.

**Solution:** Ensure that the variable and its NULL string value are desired within the statement.

*line\_number* **Variable set but not used**

**Problem:** A variable was defined but not otherwise used in the PSL program.

**Reason:** A variable was created when it was assigned a value other than the default NULL string value, but the variable was never used within the PSL program. A message that a variable was used but not set often indicates that a typographical error occurred and that PSL created two variables, one with the correct name and the other with the misspelled one.

**Solution:** Check to see why the variable was never used within the program. If it is not needed, remove it. If it is a typographical error, correct it.

*line\_number* **Undefined backslash escape in string constant**

**Problem:** An undefined escape sequence appeared in a PSL character string.

**Reason:** The correct PSL escape sequences are `\A` through `\z`, `\t`, `\n`, `\r`, and `\b`. PSL ignores and discards any other escape sequences that it detects in a character string. For example, PSL changes the string `A\zB` to the string `AB` because `\z` is not a valid escape character.

**Solution:** Verify the presence of the escape character in the character string and either modify it to be a valid PSL escape character or remove it.

*line\_number* **Float constant starts with '.'**

**Problem:** A floating point number begins with a decimal point.

**Reason:** Floating point numbers within PSL must begin with either a minus sign or a digit in order for PSL to correctly interpret them as numbers in numeric calculations. PSL will interpret a number that begins with a decimal point as a character string and will treat it as a character string in numeric operations.

**Solution:** Prefix each floating point number with 0 or a minus sign to ensure it is properly interpreted by PSL.

# Built-in Function Run-Time Error Messages

Execution of a PSL script can cause various run-time error messages from the built-in variables. These messages indicate serious PSL script problems that cannot be suppressed. This section lists the error messages and explains the most likely cause for the error.

The first word in the error message indicates the built-in function that generated the message.

## **BMCcat:** *filename* -- *error\_message*

Reason: The `cat()` function had problems opening the file such as *file not found* or *permission denied*. The system error message is reported using the PSL `errno` variable.

## **BMCcat:** *filename* -- **read error after** *number* **bytes**

Reason: The `cat()` function read operation failed after *number* bytes were read.

## **BMCcreate:** **unknown application** *application*

Reason: The `create()` function could not locate *application* in the computer symbol table.

## **BMCcreate:** **\_\_type\_\_ missing in object** *object*

Reason: The `create()` function could not find the special built-in variable `__type__` in the symbol table.

## **BMCcreate:** **object** *object* **is not an application**

Reason: The `create()` function determined that the *object* being created is not an application. The `create()` function cannot create computer instances.

**BMCcreate: \_\_self\_\_ missing in object** *object*

Reason: The create() function could not find the special built-in variable `__self__` in the symbol table.

**BMCcreate: cannot create instance** *instance* **of application** *application*

Reason: The create() function discovered that either the name *application* is bad or that too many instances are already registered for *application*.

**BMCcreate: invalid initial state** *state* **for instance** *instance* **of application** *application*

Reason: The create() function discovered that the *state* passed to it is not one of the valid states: ALARM, OK, WARN, or OFFLINE.

**BMCcreate: empty symbol table in object** *object*

Reason: The create() function discovered that the symbol table for *object* is empty, as would be the case if *object* were the parent of a computer instance. The create() function cannot create computer instances.

**BMCchange\_state: unknown object** *object*

Reason: The change\_state() function could not find *object* whose state was to change.

**BMCchange\_state: \_\_type\_\_ missing in object** *object*

Reason: The change\_state() function could not find the special built-in variable `__type__` in the symbol table.

**BMCchange\_state: object** *object* **is not an application instance**

Reason: The change\_state() function attempted to change state of a computer instance or other non-application instance object.

**BMCchange\_state: \_\_self\_\_ missing in object *object***

Reason: The change\_state() function could not find the special built-in variable `__self__` in the symbol table.

**BMCchange\_state: empty symbol table in object *object***

Reason: The change\_state() function discovered that the symbol table for *object* is empty, as would be the case if *object* were the parent of a computer instance.

**BMCdestroy: unknown object *object***

Reason: The destroy() function could not find *object* that it was to destroy.

**BMCdestroy: \_\_type\_\_ missing in object *object***

Reason: The destroy() function could not find the special built-in variable `__type__` in the symbol table.

**BMCdestroy: object *object* is not an application instance**

Reason: The destroy() function attempted to destroy a computer instance or other non-instance object. The name *object* must be an application name.

**BMCdestroy: \_\_self\_\_ missing in object *object***

Reason: The destroy() function could not find the special built-in variable `__self__` in the symbol table.

**BMCdestroy: empty symbol table in object *object***

Reason: The destroy() function discovered that the symbol table for *object* is empty, as would be the case if *object* were the parent of a computer instance.

**BMCexecute: cannot find instance** *instance*

Reason: The `execute()` command failed to find *instance* in the symbol table and so could not execute a command against it.

**BMCexecute: cannot get type of object** *object*

Reason: The `execute()` command could not find the special built-in variable `__type__` in the symbol table.

**BMCexecute: object** *object* **is not an application instance or computer**

Reason: The `execute()` command discovered that *object* is neither an application nor a computer instance and could not execute a command against it.

**BMCexecute: cannot get instance ptr of object** *object*

Reason: The `execute()` command could not find the special built-in variable `__self__` in the symbol table.

**BMCexecute: couldn't execute** *command\_type* **command**

Reason: The `execute()` command was unable to create either the internal run-time cell or the operating system process to execute the command. Possible causes include the following:

- invalid user ID
- bad application instance
- PSL program did not compile, or
- PSL process creation failed

**BMCexecute: couldn't create channel**

Reason: The `execute()` function encountered an internal PSL problem. Contact your BMC Software Product Support representative for assistance.

**BMCgrep: bad regular expression -- *regular\_expression***

Reason: The `grep()` function discovered that the regular expression argument passed to it is either not valid or too long.

**BMCget\_vars: unknown object *object***

Reason: The `get_vars()` function either:

- could not find *object*; or
- discovered that *object* does not have variables (simple variables do not have variables)

**BMCin\_transition: unknown object *object***

Reason: The `in_transition()` function could not find *object* on which it was to act.

**BMCin\_transition: \_\_type\_\_ missing in object *object***

Reason: The `in_transition()` function could not find the special built-in variable `__type__` in the symbol table.

**BMCin\_transition: object *object* is not an application instance**

Reason: The `in_transition()` function attempted to apply its timer to a computer instance or other nonapplication instance object.

**BMCin\_transition: \_\_self\_\_ missing in object *object***

Reason: The `in_transition()` function could not find the special built-in variable `__self__` in the symbol table.

**BMCin\_transition: empty symbol table in object *object***

Reason: The `in_transition()` function discovered that the symbol table for *object* is empty as would be the case if it were the parent of a computer instance.

**BMCread: bad channel#** *channel\_number*

Reason: The read() function has discovered that *channel\_number* is no longer valid for the PSL process. Possible causes include

- a bad channel number (for example, a negative number)
- a channel already closed using the close function

**BMCreadln: bad channel#** *channel\_number*

Reason: The readln() function has discovered that *channel\_number* is no longer valid for the PSL process. Possible causes include:

- a bad channel number (for example, a negative number)
- a channel already closed using the close function

**BMCwrite: bad channel#** *channel\_number*

Reason: The write() function has discovered that *channel\_number* is no longer valid for the PSL process. Possible causes include:

- a bad channel number (for example, a negative number)
- a channel already closed using the close function.

**BMCvariable: non-modifiable data type**

Reason: The set() function attempted to set *variable* in the PATROL hierarchy that has write permission but is not of a type that the set() function is allowed to modify. For example, this error occurs if *variable* is another subobject such as an application or instance.

**BMCvariable: write permission denied**

Reason: The set() function attempted to set a read-only built-in variable in the PATROL hierarchy.

---

# Internationalized PSL Scripts

This chapter discusses the built-in functions that you can use to write internationalized PSL scripts. The following topics are discussed:

Introduction . . . . .	6-2
Locale and Codeset . . . . .	6-2
Locale Categories . . . . .	6-3
set_locale() . . . . .	6-4
CTYPE Locale Category . . . . .	6-5
MESSAGES Locale Category . . . . .	6-5
CODECVT Locale Category . . . . .	6-6
TIME Locale Category . . . . .	6-7
Multiple-Byte Characters . . . . .	6-7
PSL International Functions . . . . .	6-7
ID-Based Messaging Functions . . . . .	6-8
Other PSL Functions . . . . .	6-9
Command Execution Functions . . . . .	6-9
Input and Output Functions . . . . .	6-10
File Handling Functions . . . . .	6-11
String Functions . . . . .	6-11
Set Functions . . . . .	6-12
Date and Time . . . . .	6-12
Compatibility with Noninternationalized PATROL Agents . . . . .	6-13
Example Code: Verify the Version of the PATROL Agent . . . . .	6-13
Example Code: Conditionally Use an International Function . . . . .	6-15

# Introduction

Use this chapter if you need to write PSL scripts that perform these tasks:

- Run on two or more computer platforms of different languages.
- Read and write multilingual characters and formats such as date, time, and currency.
- Exchange information between computer platforms of different languages.

An internationalized PSL script performs the tasks listed above. A noninternationalized script supports only a single language.

## Locale and Codeset

Locale is information that PATROL uses to read and write language-specific text. A locale includes information such as sort order, date and time formats, special characters, and currency formats. A locale also includes a codeset, which is a group of rules that the PATROL Agent uses to interpret the ones and zeros of a text file into characters. The eucJP codeset is an example of a Japanese codeset name.

A locale name follows the format *language\_country.codeset*. The *language* code consists of two lowercase letters that are defined by ISO 639, *country* consists of two uppercase letters that are defined by ISO 3166, and *codeset*, as previously discussed, is the name of the codeset.

Table 6-1 shows the locale names that PATROL supports.

**Table 6-1 Supported Locale Names (Part 1 of 2)**

Locale Name	Language
C	English
ja_JP.CP932	Japanese
ja_JP.eucJP	Japanese
ja_JP.SJIS	Japanese

**Table 6-1 Supported Locale Names (Part 2 of 2)**

<b>Locale Name</b>	<b>Language</b>
ko_KR.eucKR	Korean
zh_CN.gb	Simplified Chinese
zh_TW.big5	Traditional Chinese
zh_TW.eucTW	Traditional Chinese

The locale and codeset registries list additional locale and codeset names for mapping purposes. The names listed in Table 6-1 are currently the only ones that you can use with PSL functions. You can find the registries at the following directory paths:

- *PATROL\_HOME/lib/nls/locale/loc\_registry*
- *PATROL\_HOME/lib/nls/charmaps/cs\_registry*

Locale name plays an important role in writing internationalized PSL scripts. If you read, write, or display text, you must verify that you are using the right locale to process the text. For example, if your script needs to read a text file encoded with the ja\_JP.eucJP locale name, you must verify that PATROL is configured to read the file with the ja\_JP.eucJP locale.

## Locale Categories

A locale category tells PATROL which locale to use with specific functions. The TIME locale category, for example, affects the following functions: `asctime()`, `date()`, and `convert_locale_date()`. These functions are referred to as TIME functions. If you set TIME to the ja\_JP.SJIS locale, the TIME functions read, write, and display text in the ja\_JP.SJIS locale.

Table 6-2 lists the locale categories.

**Table 6-2 PSL Locale Categories**

<b>Category</b>	<b>Purpose</b>
CTYPE	<p>Specifies the locale of the agent. CTYPE functions manipulate text strings. See “CTYPE Locale Category” on page 6-5.</p> <p><b>Functions:</b> code_cvt(), grep(), index(), length(), lines(), ntharg(), nthargf(), nthline(), nthlinef(), num_bytes(), rindex(), substr(), tail(), tolower(), toupper(), trim()</p>
MESSAGES	<p>Specifies the locale that PATROL uses to display messages. MESSAGES functions display messages to the user. See “MESSAGES Locale Category” on page 6-5.</p> <p><b>Default Locale:</b> Same as CTYPE</p> <p><b>Functions:</b> msg_check(), msg_get_format(), msg_get_severity(), msg_printf(), msg_sprintf(), dcget_text(), dget_text(), get_text(), text_domain()</p>
TIME	<p>Specifies the locale of the date and time format that you want to use with a PSL function. TIME functions read, write, format, or parse a date and time string. See “TIME Locale Category” on page 6-7.</p> <p><b>Default Locale:</b> Same as CTYPE</p> <p><b>Functions:</b> asctime(), date(), convert_locale_date()</p>
CODECVT	<p>Specifies the locale of text that you want to write, read, or display. CODECVT functions execute commands and read and write text. See “CODECVT Locale Category” on page 6-6.</p> <p><b>Default Locale:</b> Same as CTYPE</p> <p><b>Functions:</b> cat(), execute(), fopen(), popen(), read(), readln(), system, write()</p>

## set\_locale()

The set\_locale() function is the tool that you use to check and change the locale of a locale category. For example, asctime() can return a date or time in any locale. If you want it to return the date in the ja\_JP.eucJP locale, use set\_locale() to verify the TIME locale category. If set\_locale() says that TIME is set to ja\_JP.SJIS, use set\_locale() again to change the locale of TIME to ja\_JP.eucJP. For more information about set\_locale(), see *PATROL Script Language Reference Manual Volume 2—PSL Functions*.

## CTYPE Locale Category

The CTYPE locale category specifies the native locale of the PATROL Agent. You can use `set_locale()` to read the locale of CTYPE, but you cannot change it. CTYPE is the only nonwritable locale category.

The agent can process only text that is in the locale of the CTYPE locale category so all string functions follow the rules of the CTYPE locale. Using the `sort()` function, for example, sorts a list of strings as they exist in the CTYPE locale.

The value of CTYPE also serves as the default value of the other locale categories. When a PSL script executes, the initial value of the other locale categories is equal to the value of CTYPE.

---

### Note

---

CTYPE PSL functions support multiple-byte characters. See “Multiple-Byte Characters” on page 6-7.

---

## MESSAGES Locale Category

The MESSAGES locale category specifies the locale that PATROL uses to display messages to the user. An internationalized PSL script does not display messages directly to the user. Instead, the script has a message catalog for each locale that it supports. When the script displays a message, it uses an ID number to retrieve the message string from a catalog and displays the message to the user. The locale of MESSAGES tells PATROL which message catalog to use. For example, if the user reads the Chinese Simplified locale, the script verifies whether MESSAGES is set for the Chinese Simplified locale before displaying any messages.

## CODECVT Locale Category

The CODECVT locale category specifies the locale of input and output text. Input text is text that PATROL reads from a channel or file, but it is also text that a command returns. Output text is text that PATROL writes to a channel or file.

### Codeset Conversion

PATROL converts the codeset of any text identified by the CODECVT locale category if it does not match the locale of the agent (or CTYPE locale category). The conversion process is nearly automatic. The only step that you must perform in the conversion process is to verify that the CODECVT locale category is set correctly before using a CODECVT function.

---

#### Note

---

You can only convert between codesets that represent the same language such as converting between ja\_JP.eucJP and ja\_JP.SJIS for Japanese or between zh\_TW.big5 and zh\_TW.eucTW for Chinese.

---

For example, say that a PSL script exists on a different computer from the PATROL Agent. The operating system that hosts the script uses the ja\_JP.SJIS locale, and the operating system for the agent uses ja\_JP.eucJP. When the script executes the `system()` function, it returns an operating system message in the ja\_JP.SJIS locale. In this example, you need to verify that the CODECVT locale category is set for ja\_JP.SJIS before using `system()`. PATROL automatically converts the message string to the CTYPE or ja\_JP.eucJP locale so that the agent can read the message.

## Registered Locale for a Channel

The `popen()` and `fopen()` functions play a special role in locale conversion. These functions open a channel through which *PATROL* can input and output text. When you use `popen()` or `fopen()` to open a channel, the locale of the `CODECVT` locale category becomes the *registered* locale for the channel. A registered locale means that you cannot change the locale of the channel, but you can close the channel and open a new one with a different registered locale. If the registered locale is different from the `CTYPE` locale, *PATROL* automatically converts text during read and write operations to the channel.

## TIME Locale Category

The `TIME` locale category specifies the date and time format of a text string that is returned from one of the following PSL functions: `asctime()`, `date()`, and `convert_locale_date()`.

## Multiple-Byte Characters

Generally, English language codesets use a single byte to represent a character. Some international codesets need more than one byte to represent a single character because many international alphabets have a very large number of characters. Any character that requires more than one byte is a multiple-byte character. All PSL built-in string functions support multiple-byte characters. You can, for example, use multiple-byte characters in the following types of string operations: string comparisons, sorts, and range expressions using regular expressions.

## PSL International Functions

The functions in Table 6-3 can help you write internationalized PSL scripts. For more information about any PSL function, see *PATROL Script Language Reference Manual Volume 2—PSL Functions*.

**Table 6-3 PSL International Functions**

<b>Function</b>	<b>Definition</b>
<b>General International Functions</b>	
code_cvt() convert_locale_date() num_bytes() set_locale()	convert a string from one codeset to another change the format of a date string return the length of a string in bytes set or get the value of a locale category
<b>ID-Based Messaging Functions</b>	
msg_check() msg_get_format() msg_get_severity() msg_printf() msg_sprintf()	check the validity of a message ID and message catalog return the format information for a message return a integer that explains the severity of a message print message text formatted by information from a catalog return message text formatted by information from a catalog
<b>String-Based Messaging Functions</b>	
dcget_text() dget_text() get_text() text_domain()	return message text by defining domain and category return message text by defining only domain return message text set or get the current name of the domain

## **ID-Based Messaging Functions**

The ID-based messaging functions in Table 6-3 require that you use a message catalog. The following PATROL manuals describe how to create a message catalog: *PATROL Console for Unix User Guide* and *PATROL Console for Microsoft Windows User Guide Customizing PATROL Volume 3*.

To use an existing ID-based message catalog, you must know the name of the **.lib** file for the catalog. In internationalized scripts, use the `requires` statement to identify the **.lib** file and call an INIT function based on the **.lib** name. For example, if the name of the **.lib** is **example.lib**, include the following lines of code:

```
requires "example.lib";
. . .
INIT_example();
```

The example catalog contains the following information:

```
VENDOR_ID 1
VENDOR    BMC Software
TOOL_ID   23
TOOL      PATROL Demo
VERSION   1.0
DATE      13-Mar-1997
#-----
ID        145          DEMO_HELLO          INFO          CURRENT
MESSAGE  "Hello World"
```

The `requires` statement makes the message catalog available to your script. The `INIT` function allows you to reference either the message ID number or message ID name. Without this function, you can use only the ID number. In the preceding example, executing the `INIT` function makes the `msg_check(DEMO_HELLO)` command equal to `msg_check(145)`.

## Other PSL Functions

Unlike the functions in Table 6-3 on page 6-8, the functions described in this section existed before internationalized versions of PATROL were available. These functions have little to do with internationalized PSL development, but each function has a feature or two that supports internationalization. This section briefly describes these features.

## Command Execution Functions

Table 6-4 lists the command execution functions that have internationalization features.

**Table 6-4 International Features of Command Execution Functions**

Function	Description
execute()	You should set the CODECVT locale category to match the locale of the command output. If CODECVT differs from CTYPE, this function converts the locale and returns text in CTYPE.
fopen()	You should set the CODECVT locale category to match the locale of the text data in the file. This function opens a channel that has a registered locale equal to CODECVT. During subsequent read and write operations, PATROL automatically converts the locale when necessary.
popen()	You should set the CODECVT locale category to match the locale of the text data in the channel. This function opens a channel that has a registered locale equal to CODECVT. During subsequent read and write operations, PATROL automatically converts the locale when necessary.
system()	You should set the CODECVT locale category to match the locale of the command output. If CODECVT differs from CTYPE, this function converts the locale and returns text in CTYPE.

## Input and Output Functions

Table 6-5 lists the input and output functions that have internationalization features.

**Table 6-5 International Features of Input and Output Functions**

Function	Description
printf() sprintf()	The printf() and sprintf() functions support specification of the converted argument position.
read()	If the registered locale for the channel differs from CTYPE, read() converts the locale and returns text in the CTYPE locale. The popen() and fopen() functions register the locale for a channel. The size parameter specifies the number of bytes that the function reads from the channel. <b>Note:</b> This description differs from the PATROL 3.2.09i version of read().
readln()	If the registered locale for the channel differs from CTYPE, readln() converts the locale and returns text in the CTYPE locale. The popen() and fopen() functions register the locale for a channel.
write()	If the registered locale for the channel differs from CTYPE, write() converts the locale and writes text data to the channel in the locale for the channel. The popen() and fopen() functions register the locale for a channel.

# File Handling Functions

Table 6-6 lists the file-handling functions that have internationalization features.

**Table 6-6 International Features of File Handling Functions**

Function	Description
cat()	You should set the CODECVT locale category to match the locale of the text data in the file. If CODECVT differs from CTYPE, this function converts the locale and returns text in CTYPE.
fseek()	The <i>offset</i> parameter of the <code>fseek()</code> function specifies the number of bytes. <b>Note:</b> In PATROL 3.2.09i, <i>offset</i> specifies the number of characters (including multiple-byte characters).
ftell()	The <code>ftell()</code> function returns file position as the number of bytes from the beginning of the file. <b>Note:</b> In PATROL 3.2.09i, this function returns the number of characters (including multiple-byte characters).

# String Functions

Table 6-7 lists the string functions that have internationalization features.

**Table 6-7 International Features of String Functions (Part 1 of 2)**

Function	Description
grep()	The <code>grep()</code> function supports multiple-byte characters in the <i>regular-expression</i> and <i>text</i> parameters. It also supports ranges expression using code point ordering.
index() rindex()	The <code>index()</code> and <code>rindex()</code> functions support multiple-byte characters in the <i>text</i> and <i>string</i> parameters. These functions return position counts in characters.
length()	The <code>length()</code> function returns the number of characters (including multiple-byte characters).
lines() nthline() nthlinef() tail()	These functions support multiple-byte characters in the <i>text</i> and <i>separator</i> parameters.
ntharg() nthargf()	The <code>ntharg()</code> and <code>nthargf()</code> functions support multiple-byte characters in the <i>text</i> , <i>delimiters</i> and <i>separator</i> parameters.

**Table 6-7 International Features of String Functions (Part 2 of 2)**

<b>Function</b>	<b>Description</b>
substr()	The substr() function supports multiple-byte characters in the <i>text</i> parameter. The <i>start</i> and <i>length</i> parameters specify the number of characters.
tolower() toupper()	The tolower() and toupper() functions use the locale of the CTYPE locale category to change the case of text.
trim()	The trim() function supports multiple-byte characters in the <i>str</i> and <i>chars</i> parameters.

## Set Functions

Table 6-8 lists the set functions that have internationalization features.

**Table 6-8 International Features of Set Functions**

<b>Function</b>	<b>Description</b>
sort()	The sort() function supports multiple-byte characters in <i>list</i> element strings using code point ordering. The <i>position</i> parameter specifies the position in terms of the number of characters from the beginning of the list.
difference() intersection() subset() union() unique()	These functions support multiple-byte characters in the element strings.

## Date and Time

Table 6-9 lists the date and time functions that have internationalization features.

**Table 6-9 International Features of Date and Time Functions**

<b>Function</b>	<b>Description</b>
asctime() date()	The asctime() and date() functions use a default date and time format that is determined by the locale of the TIME locale category.

# Compatibility with Noninternationalized PATROL Agents

A PATROL Agent with a version number prior to 3.4.11 does not support international PSL functions (Table 6-3 on page 6-8). Before using these functions, you must verify that the version of the agent is 3.4.11 or later. You can use the examples in this section to create scripts that use international PSL functions in a way that is compatible with noninternationalized versions of the agent.

## Example Code: Verify the Version of the PATROL Agent

The example code in this section creates the `PatrolVersionCheck()` function, which evaluates the version number of the PATROL Agent. The `PatrolVersionCheck()` function returns a value greater than 0 if the version number indicates that the agent can run international PSL functions. Otherwise, it returns a 0.

```

export function i18n_set_locale;
function PatrolVersionCheck()
{
i18n_version = 0;
patrol_version = substr(get("/patrolVersion"), 2, 1);
patrol_release = ntharg(get("/patrolVersion"), 2,
".");
patrol_release2 = ntharg(get("/patrolVersion"), 3,
".");
patrol_release3 = ntharg(get("/patrolVersion"), 4,
".");

if( patrol_version == 3 )
{
    if (( patrol_release == 4 ) && (patrol_release2 >=
11))
    {
        i18n_version = 1;
    }
    if (patrol_release > 4)
    {
        i18n_version = 1;
    }
    if ((patrol_release == 2) && (patrol_release2 == 9)
&&
        (patrol_release3 >=1))
    {
        i18n_version = 2;
    }
}

if( patrol_version > 3)
{
    i18n_version = 1;
}

return i18n_version;
}

```

## Example Code: Conditionally Use an International Function

The example code in this section creates the `i18n_set_locale()` function to provide a way of using `set_locale()` that is compatible with noninternationalized versions of the PATROL Agent. The `i18n_set_locale()` function determines whether the agent can run international PSL functions by using the `PatrolVersionCheck()` function (“Example Code: Verify the Version of the PATROL Agent” on page 6-13). If the agent can run these functions, it runs the `set_locale()` function. Otherwise, it returns a NULL string.

```
function i18n_set_locale(cat,value)
{
    if (PatrolVersionCheck() >0 )
    {
        return set_locale(cat,value);
    }
    else
    {
        return "";
    }
}
```



## errno Return Values

This appendix lists the values returned by the PSL `errno` variable. Table A-1 lists the numeric return value of the `errno` variable and its corresponding message text.

**Table A-1 PSL errno Values (Part 1 of 6)**

<b>Numeric Value</b>	<b>Message</b>
0	E_PSL_NO_ERROR
1	E_PSL_CAT_MEMORY_FAILURE
2	E_PSL_CAT_READ_ERROR
3	E_PSL_CHANGE_STATE_EMPTY_SYMBOL_TABLE
4	E_PSL_CHANGE_STATE_INTERNAL_FAILURE
5	E_PSL_CHANGE_STATE_INVALID_STATE
6	E_PSL_CHANGE_STATE_NOT_APPLICATION
7	E_PSL_CHANGE_STATE_UNKNOWN_OBJECT
8	E_PSL_CLOSE_BAD_CHANNEL
9	E_PSL_CREATE_ALREADY_EXISTS
10	E_PSL_CREATE_BAD_STATE
11	E_PSL_CREATE_CANNOT_CREATE
12	E_PSL_CREATE_EMPTY
13	E_PSL_CREATE_EMPTY_SYMBOL_TABLE
14	E_PSL_CREATE_INTERNAL_FAILURE
15	E_PSL_CREATE_INACTIVE_APPLICATION

**Table A-1 PSL errno Values (Part 2 of 6)**

<b>Numeric Value</b>	<b>Message</b>
16	E_PSL_CREATE_NOT_APPLICATION
17	E_PSL_CREATE_UNKNOWN_APPLICATION
18	E_PSL_CREATE_SUPPRESSED
19	E_PSL_DESTROY_EMPTY_SYMBOL_TABLE
20	E_PSL_DESTROY_INTERNAL_FAILURE
21	E_PSL_DESTROY_NOT_APPLICATION
22	E_PSL_DESTROY_UNKNOWN_OBJECT
23	E_PSL_EXECUTE_BAD_INSTANCE
24	E_PSL_EXECUTE_CANNOT_CREATE_CHANNEL
25	E_PSL_EXECUTE_CANNOT_EXECUTE
26	E_PSL_EXECUTE_INTERNAL_FAILURE
27	E_PSL_EXECUTE_NOT_IMPLEMENTED
28	E_PSL_EXECUTE_NOT_INSTANCE
29	E_PSL_FILE_CANNOT_OPEN
30	E_PSL_FILE_NOT_FOUND
31	E_PSL_FILE_READ_ERROR
32	E_PSL_FILE_SECURITY_FAILED
33	E_PSL_FILE_STAT_FAILED
34	E_PSL_FOPEN_BAD_MODE
35	E_PSL_FOPEN_CANNOT_CREATE_CHANNEL
36	E_PSL_FOPEN_CANNOT_OPEN_FILE
37	E_PSL_FOPEN_FILE_NOT_FOUND
38	E_PSL_FOPEN_STAT_FAILED
39	E_PSL_GETENV_NOT_FOUND
40	E_PSL_GET_CHAN_INFO_BAD_CHANNEL
41	E_PSL_GET_NOT_FOUND
42	E_PSL_GET_OBJECT
43	E_PSL_GET_VARS
44	E_PSL_GREP_BAD_REGEX

**Table A-1 PSL errno Values (Part 3 of 6)**

<b>Numeric Value</b>	<b>Message</b>
45	E_PSL_IN_TRANSITION_EMPTY_SYMBOL_TABLE
46	E_PSL_IN_TRANSITION_INTERNAL_FAILURE
47	E_PSL_IN_TRANSITION_NOT_APPLICATION
48	E_PSL_IN_TRANSITION_UNKNOWN_OBJECT
49	E_PSL_PRINTF_FORMAT
50	E_PSL_PRINTF_STAR
51	E_PSL_PRINTF_TOO_FEW
52	E_PSL_PROC_EXISTS_BAD
53	E_PSL_READLN_BAD_CHANNEL
54	E_PSL_READ_BAD_CHANNEL
55	E_PSL_READ_FAILED
56	E_PSL_READ_FAILED_MEMORY
57	E_PSL_READLN_TRUNCATED
58	E_PSL_SET_NON_MODIFIABLE
59	E_PSL_SET_SYMBOL_TABLE
60	E_PSL_SET_WRITE_PERMISSION
61	E_PSL_WRITE_BAD_CHANNEL
62	E_PSL_WRITE_CHANNEL_CLOSED
63	E_PSL_WRITE_FAILED
64	E_PSL_WRITE_READ_ONLY
65	E_PSL_BAD_UNLOCK
66	E_PSL_BAD_UNLOCK_NOT_OURS
67	E_PSL_BAD_LOCK_WAITING
68	E_PSL_EDOM
69	E_PSL_ERANGE
70	E_PSL_FOPEN_BAD_CHMOD
71	E_PSL_FOPEN_BAD_CHOWN
72	E_PSL_FOPEN_BAD_ACCOUNT
73	E_PSL_PRINTF_LARGE_ARGUMENT

**Table A-1 PSL errno Values (Part 4 of 6)**

<b>Numeric Value</b>	<b>Message</b>
74	E_PSL_SORT_BAD_MODE
75	E_PSL_HISTORY_BAD_OBJ
76	E_PSL_HISTORY_NOT_PARAM
77	E_PSL_HISTORY_NO_RTCELL
78	E_PSL_HISTORY_NO_INFO
79	E_PSL_HISTORY_BAD_FORMAT
80	E_PSL_FSEEK_PIPE_CHANNEL
81	E_PSL_FSEEK_BAD_CHANNEL
82	E_PSL_FTELL_PIPE_CHANNEL
83	E_PSL_FTELL_BAD_CHANNEL
84	E_PSL_RESPONSE_NO_VALUE
85	E_PSL_RESPONSE_NO_CONSOLE
86	E_PSL_RESPONSE_TIMEOUT
87	E_PSL_SHARE_BAD_CHANNEL
88	E_PSL_SHARE_SAME_CHANNEL
89	E_PSL_CLOSE_BUSY_CHANNEL
90	E_PSL_BUSY_CHANNEL
91	E_PSL_UNBLOCKED_BY_CLOSE
92	E_PSL_FULL_DISCOVERY_BAD_APP
93	E_PSL_NO_SUCH_ID
94	E_PSL_SOCKET_BUSY
95	E_PSL_TIMEOUT
96	E_PSL_BAD_FUNCTION_PARAMETER
97	E_PSL_SNMP_NOT_SUPPORTED
98	E_PSL_SNMP_ALREADY_LISTENING
99	E_PSL_SNMP_NOT_LISTENING
100	E_PSL_SNMP_ERROR
101	E_PSL_BAD_OBJ
102	E_PSL_NOT_PARAM

**Table A-1 PSL errno Values (Part 5 of 6)**

<b>Numeric Value</b>	<b>Message</b>
103	E_PSL_NO_RTCELL
104	E_PSL_NOT_SUPPORTED
105	E_PSL_ANN_NO_HISTORY_RETENTION
106	E_PSL_REMOTE_QUERY_COMM_ERR
107	E_PSL_REMOTE_QUERY_CREATION
108	E_PSL_REMOTE_QUERY_CANNOT_START
109	E_PSL_REMOTE_OPEN_ERR
110	E_PSL_FAULT
111	E_PSL_BAD_DATE_STRING
112	E_PSL_BAD_TIMEZONE
113	E_PSL_CANNOT_DETERMINE_TIMEZONE
114	E_PSL_LOCK_DESTROYED
115	E_PSL_ANN_TOO_EARLY
116	E_PSL_ANN_SAVE_FAILED
117	E_PSL_ACL_FAILED
118	E_PSL_PCONFIG_FAILED
119	E_PSL_NO_ANNOTATION_DATA
120	E_PSL_BAD_CHART_COMMAND
121	E_PSL_ALLOC_FAILED
122	E_PSL_INTERNAL_ARG_ERROR
123	E_PSL_INTERNAL_PARSE_FAILED
124	E_PSL_INTERNAL_UNKNOWN_FUNCTION
125	E_PSL_INTERNAL_FUNCTION_FAILED
126	E_PSL_INTERNAL_TYPE_MISMATCH
127	E_PSL_INTERNAL_BAD_NAME
128	E_PSL_ALREADY_WAITING
129	E_PSL_ENABLED
130	E_PSL_DISABLED
131	E_PSL_BAD_METRIC_GROUP

**Table A-1 PSL errno Values (Part 6 of 6)**

<b>Numeric Value</b>	<b>Message</b>
132	E_PSL_BAD_METRIC
133	E_PSL_BAD_COMPUTATION
134	E_PSL_SET_LOCALE_FAILED
135	E_PSL_CODECVT_INVALID_MB
136	E_PSL_CODECVT_NO_CONV
137	E_PSL_CODECVT_SAME
138	E_PSL_RTE_FSEEK_RESTRICTED_FILEOPEN_MODE
139	E_PSL_SKS_BAD_ACCOUNT
140	E_PSL_SKS_INSTANCE_ERR
141	E_PSL_SKS_SET_ERR
142	E_PSL_SKS_GET_ERR

---

# Built-in Agent Namespace Variables

The built-in Agent namespace variables maintained by the PatrolServer are presented here in table format. You can use these variables to customize existing commands and parameters. The tables define:

Computer Class Built-in Variables . . . . .	B-2
Application Class Built-in Variables . . . . .	B-3
Application Instance Built-in Variables . . . . .	B-4
Parameter Built-in Variables . . . . .	B-5

The Mode column in each table shows *r-* for read-only variables, and *rw* for read-write variables.

# Computer Class Built-in Variables

**Table B-1 Computer Class Built-in Variables (Part 1 of 2)**

<b>Mode</b>	<b>Name</b>	<b>Description</b>
r-	objectId	Object ID
r-	appType	Application name (computer class)
r-	sid	Host name
r-	name	Host name
r-	hostname	Host name
r-	ipAddress	Host IP address
r-	tcpPort	TCP port to which PatrolServer is bound
r-	patrolHome	PATROL home directory
r-	patrolVersion	Version of the PatrolServer software
r-	ruleState	Status of the instance based on discovery information only
r-	worstParam	Name of “worst” parameter on this computer
r-	worstParamState	Status of “worst” parameter on this computer
r-	status	Overall status of this instance (“worst” of ruleState, worstParamState and worstApplInstState)
r-	globalParamsSuspended	TRUE if global parameters have been suspended for this computer, FALSE otherwise
r-	execParams	TRUE if parameters for this computer are allowed to be executed, FALSE otherwise
r-	agentPid	PID of the PatrolServer process
r-	agentLogPath	Full path name of the PatrolServer's error log file
r-	serverPid	PID of the PatrolServer's server process
r-	serverLogPath	Full path name of the server's error log file
r-	totalErrors	Total number of errors detected by the PatrolServer
r-	allocErrors	Number of memory allocation errors
r-	forkErrors	Number of process spawning errors
r-	pipeErrors	Number of pipe creation errors
r-	internalErrors	Number of miscellaneous internal errors
r-	userErrors	Number of errors in user-specified commands

**Table B-1 Computer Class Built-in Variables (Part 2 of 2)**

<b>Mode</b>	<b>Name</b>	<b>Description</b>
r-	executingProcs	Number of currently executing sub-processes (parameters, commands, and so on)
r-	execsPerMin	Average number of sub-processes started per minute
r-	timeBtnExecs	Average time (in seconds) between spawning of sub-processes
r-	time	Current time as ASCII string (e.g., Thu May 07 02:44:01 1992)

## Application Class Built-in Variables

**Table B-2 Application Class Built-in Variables**

<b>Mode</b>	<b>Name</b>	<b>Description</b>
r-	.	Symbol table for application
r-	..	Symbol table for computer
r-	name	Application name
rw	active	TRUE if discovery for application is active, FALSE otherwise
r-	lastDiscoveryTime	Time (in seconds from Epoch) at which discovery of this application was last done
r-	username	Application user name
r-	propagateState	TRUE if the state of this application is automatically propagated to the computer
r-	environment	Environment variables
r-	instances	List of SIDs of instances of this application
r-	numInstances	Number of instances of this application

# Application Instance Built-in Variables

Table B-3 Application Instance Built-in Variables (Part 1 of 2)

Mode	Name	Description
r-	.	Symbol table for instance
r-	..	Symbol table for application
r-	objectId	Object ID
r-	appType	Application name
r-	sid	SID for instance
rw	name	Icon name/label for instance
r-	username	Application user name
rw	environment	Environment variables
rw	home	Home directory for instance
rw	version	Instance version (software)
r-	ruleState	Status of the instance based on discovery information only
r-	worstParam	Name of “worst” parameter on this instance
r-	worstParamState	Status of “worst” parameter on this instance
r-	status	Overall status of this instance (“worst” of ruleState and worstParamState)
r-	globalParamsSuspended	TRUE if global parameters have been suspended for this instance, FALSE otherwise
r-	execParams	TRUE if parameters for this instance are allowed to be executed, FALSE otherwise
r-	transition	Time (in seconds from Epoch) at which instance entered state transition, or zero if not in transition
r-	procUser	User names of processes belonging to this instance
r-	procPid	PIDs of processes belonging to this instance
r-	procParentPid	Parent PIDs of processes belonging to this instance
r-	procName	Names of processes belonging to this instance
r-	procCommand	Full commands of processes belonging to this instance
r-	procSize	Sizes (virtual) of processes belonging to this instance
r-	procResidentSetSize	Sizes (resident) of processes belonging to this instance

**Table B-3 Application Instance Built-in Variables (Part 2 of 2)**

<b>Mode</b>	<b>Name</b>	<b>Description</b>
r-	procStatus	States of processes belonging to this instance
r-	procPercentCpu	CPU utilization for processes belonging to this instance
r-	procCpuTime	Cumulative CPU consumption for processes belonging to this instance
r-	parentInstance	Path to the nested instance's parent instance

## Parameter Built-in Variables

**Table B-4 Parameter Built-in Variables**

<b>Mode</b>	<b>Name</b>	<b>Description</b>
r-	.	Symbol table for parameter
r-	..	Symbol table for instance/computer
r-	name	Parameter name
rw	active	Parameter activity
rw	status	State of the parameter
r-	execTime	Time of next scheduled execution
rw	interval	Interval (in seconds) between executions
rw	value	Current value
r-	alarmMin	Minimum value of current alarm's range, or zero if no alarm is active
r-	alarmMax	Maximum value of current alarm's range, or zero if no alarm is active
r-	alarmState	State of current alarm, or "No alarm active" if no alarm is active



---

## Additional PSL Tools

This appendix describes additional PSL tools you can use when writing PSL scripts. The additional PSL tools are

PSL Profiler Tool . . . . .	C-2
How to Install the PSL Profiler . . . . .	C-4
How to Start the PSL Profiler . . . . .	C-4
PSL Profiler PSL Functions . . . . .	C-5
About the PSL Profile Viewer (ppv) Tool . . . . .	C-8
About the PSL Profiler API . . . . .	C-8
PSL Optimizer Tool . . . . .	C-10
Introduction to the PSL Optimizer . . . . .	C-10
How to Install the PSL Optimizer . . . . .	C-10
How to Deactivate the PSL Optimizer . . . . .	C-11
About the PSL Optimizer . . . . .	C-11
Optimization Levels . . . . .	C-11
Optimization Criteria . . . . .	C-14
Command-Line Specified Options . . . . .	C-16

# PSL Profiler Tool

The PSL Profiler is a measurement component built into the PATROL Agent to measure how much resources each PSL script consumes. This allows the determination of which scripts have the largest usage cost and provides information as to where the cost is distributed within the PSL script.

The main PATROL Agent resource measured by the PSL Profiler is CPU time. This is broken into user time and system time, to indicate that the system calls may take some time in some of the PSL built-in functions. The PSL Profiler does not explicitly measure other cost measures such as the number of child processes, the amount of memory, PSL locks, global channels, and so on. However, some of this information can be gathered from the execution counts and the built-in function call counts. For example, child processes can be measured reasonably accurately as the number of calls to PSL `system()`, `execute()`, and `popen()` functions.

The PSL Profiler tool is used to collect profiling information from PSL processes. Primarily, the PSL Profiler records which PSL functions are called, how often each PSL function is called, and the time spent in each PSL function for each PSL process. All the profiling information that is collected for a PSL process is stored internally until the PSL process exits. When the PSL process does exit, the profiling information is written to the binary file specified by the `PSL_PROF_LOG` environment variable.

The PSL Profiler saves the profile data for PSL processes to the profile file as they terminate. Profiler output is saved in the file specified by the `PSL_PROF_LOG` environment variable. When the PATROL Agent is shut down in an orderly fashion, the Profiler saves the profile data for those PSL processes that have not been destroyed.

There are a number of interfaces to the PSL Profiler:

- **PSL functions**—The PSL Profiler can be dynamically enabled via PSL and reports can be printed in PSL.

- **PSL Profiler Viewer (ppv) utility**—This is an off-line reporting tool that analyzes a profiling data file which the PATROL Agent can generate during profiling.
- **PSL Profiler API**—The PSL Profiler API allows the PSL Profiler to be controlled (started, stopped, etc.) and queried at run-time.

## Supported Platforms

### Operating Systems

- UNIX
- Windows
- OS/2
- VMS

### PATROL Versions

PATROL 3.2 and later Agent

### To Use the PSL Profiling Capability in PATROL v3.1

The list of PSL processes currently running in the Agent can be sampled and used to generate a rough picture of the cost of each PSL process. Use this tool as a sampling technique for PATROL v3.1 products in lieu of the PSL Profiler offered in v3.2.

## Resource Requirements

PSL Profiler requires no additional resources beyond those specified for running the PATROL for Windows product.

## When to Use the PSL Profiler

The primary purpose of the PATROL profiling tools in the KM development environment is to aid in the KM performance tuning task.

Use the PSL Profiler to

- tune the CPU usage of a PSL script

- find the highest consuming PSL script
- tune the number of external processes

The Profiler tools can also be used as a PATROL Agent configuration or deployment tool in production systems to determine which KMs and PSL scripts are taking too many resources.

### **Limitations of the PSL Profiler**

The main limitation of the PSL Profiler is that it does not go very deep within each script to examine performance. For example, there is no line-by-line report showing which lines are executed most frequently. The analysis of built-in functions is useful but there is no similar analysis of PSL user-defined functions. The PSL Profiler can be dynamically enabled and disabled during Agent execution. Enabling the PSL Profiler will turn on the measurement for the specified PSL scripts. Disabling the Profiler will discard all measured data and return the Agent to the normal non-profiling state.

The impact of turning on the PSL Profiler is a marginal degradation of PSL execution performance. This results because each PSL timeslice must also be added to measurement counters. However, this is mainly a low in-memory cost. The largest costs are when the reports are generated. Report generation occurs only when the data has been collected.

## **How to Install the PSL Profiler**

This tool is part of the PATROL product. No further setup is required once PATROL has been installed.

## **How to Start the PSL Profiler**

You can activate the PSL Profiler:

- using PSL functions
- starting the PATROL Agent with the `-profiling` option

In all cases, the output report is of a similar format. The PSL Profiler reports tell you which PSL scripts have the highest CPU usage and also measure how much each built-in PSL function has used.

Scheduling frequency of the PSL processes affects the results since the CPU usage timers are cumulative. However, since the number of executions is also measured, the average cost per execution can also be easily generated in reports.

Profiler output is saved in the file specified by the `PSL_PROF_LOG` environment variable.

## PSL Profiler PSL Functions

The following PSL functions support the PSL Profiler tool:

Name	Description
<code>ProfDefaultOptions()</code>	sets the default Profiler options to <i>options</i>
<code>ProfGet()</code>	retrieves the profile of the process identified by <i>pid</i>
<code>ProfGetTotalCpu()</code>	retrieves the total real, user, and CPU time as well as the percent of CPU usage
<code>ProfOptions()</code>	sets the profiling options of the process identified by <i>pid</i> to <i>options</i>
<code>ProfReset()</code>	resets the profiling data for all processes
<code>ProfTop()</code>	retrieves the profiles of the <i>top_procs</i> highest CPU time processes

### ProfDefaultOptions()

The default Profiler options are queried or changed by using the `ProfDefaultOptions()` function, with the following format:

```
ProfDefaultOptions(options)
```

This function sets the default profiler options to *options*. If *options* is omitted, then the current default options are returned. Changing the default Profiler options does not affect the profiling options of existing processes.

## ProfGet()

The profiles of one or more PSL processes are retrieved by using the `ProfGet()` function, with the following format:

```
ProfGet(pid, top_funcs)
```

This function retrieves the profile of the process identified by *pid*. If *pid* is `-1` or `""`, then the profiles of all existing PSL processes are returned. If *top\_funcs*, which is a numeric value, is specified, then only *top\_funcs* highest CPU-time function calls are returned to the profile of each process. Otherwise, all function call data available for each process is returned.

## ProfGetTotalCpu()

The total CPU usage statistics for the current interval (since profiling was started or reset) are retrieved by the `ProfGetTotalCpu()` function, with the following format:

```
ProfGetTotalCpu()
```

The `ProfGetTotalCpu()` function returns, in order, the real CPU time, user CPU time, system CPU time, and CPU percent in the following format:

```
2:25:52.275  0:01:07.998  0:00:14.200  0.94
```

---

### Note

---

OS/2 does not support this function. The return values for this function will always be zero on OS/2.

---

## ProfOptions()

The profiling options of existing processes are queried or changed by using the `ProfOptions()` function, with the following format:

```
ProfOptions(pid,options)
```

This function sets the profiling options of the process identified by *pid* to *options*. If *pid* is -1 or "", then the profiling options of all existing processes are changed to *options*. If *options* is omitted, then the current profiling options of the process identified by *pid* are returned.

## ProfReset()

The profiling data for all processes are reset by using the `ProfReset()` function, with the following format:

```
ProfReset()
```

This function resets the profiling data of all processes and sets the profiler's reference point to the current time so the future `ProfGetTotalCpu()` function call reflects the totals from the `ProfReset()` function call.

## ProfTop()

The profiles of the highest CPU time-consuming processes are retrieved by using the `ProfTop()` function, with the following format:

```
ProfTop(top_procs, [top_funcs])
```

This function retrieves the profiles of the *top\_procs* highest CPU-time processes. If *top\_funcs* is specified, then the *top\_funcs* highest CPU-time functions calls are returned in the profile of each process. Otherwise, all function call data available for each process is returned.

---

### Note

---

The *top\_procs* and *top\_funcs* variables are both numeric values.

---

## About the PSL Profile Viewer (ppv) Tool

The PSL Profile Viewer tool (also known as the ppv utility) provides the capability to view the PSL profiling information stored in a binary profile.

To view the binary profile file after the Agent (or PSL stand-alone) has terminated, use the ppv utility. This tool uses one argument, the name of the profile file, and stores statistics for the PATROL Agent and each PSL process that the Agent has executed. The Agent's data includes:

- the number of PSL processes profiled
- the elapsed time of the Agent
- the cumulative CPU time of the PATROL Agent
- the average percent CPU load of the PATROL Agent

The PSL process data includes the number of executions and real and CPU times used by each process, the number of calls, and a breakdown of the real and CPU time used by each function that a process calls.

All of this information is sorted in descending order by the highest CPU time and by the name in the event that two processes (or function calls) have the same cumulative CPU time.

## About the PSL Profiler API

The PSL Profiler API allows the PSL Profiler to be controlled (started, stopped, etc.) and queried at run-time. Profiling is possible without needing the `-profiling` option and without stopping the Agent.

Profiling behavior is controlled by assigning a process the sum of one or more of the following options:

Value	Description
0	no profiling
1	process-level profiling
2	function-level profiling

Value	Description
4	cumulative
8	save at exit

When first created, a process is assigned the current default Profiler options. The sum of the default Profiler options is 15 if the Agent is started with the `-profiling` option (`-p` option for the PSL stand-alone) or 0 otherwise.

---

**Note**

---

To stop profiling, shutdown the PATROL Agent to write the data to the file.

---

# PSL Optimizer Tool

## Introduction to the PSL Optimizer

The PATROL PSL Optimizer tool is a three-level, multi-pass, intermediate code (quad) optimizer.

### Supported Platforms

#### Operating Systems

- UNIX
- Windows
- OS/2
- VMS

#### PATROL Versions

- PATROL 3.2 only

### Resource Requirements

The PSL Optimizer requires no additional resources beyond those specified for running the PATROL product.

### When to Use the PSL Optimizer

Use this tool to optimize your KM code.

## How to Install the PSL Optimizer

This tool is part of the PATROL Agent. No further setup is required once PATROL has been installed.

# How to Deactivate the PSL Optimizer

Since this tool is part of the PATROL product, there is no requirement to deactivate the tool.

## About the PSL Optimizer

When a PSL script is compiled, it is compiled down to an intermediate code. The PATROL Optimizer transforms the intermediate code generated by the PSL Compiler so that the code executes more efficiently in the PSL interpreter.

## Optimization Levels

You can specify the level of optimization by using the -O flag with the PSL stand-alone compiler or with the PatrolAgent function. You can also use the pragma keyword in PSL to set the optimization level for a particular PSL script.

The supported levels of optimization are:

Level Code	Description
0	no optimization
1	peephole optimization (default)
2	local optimization
3	global optimization

### Level 1: Peephole Optimizations

Level 1 optimizations are ignorant of a program's control flow and are limited to the analysis of a program on an instruction by instruction basis.

Level 1 optimizations include

- jump chain reduction
- useless jump removal
- redundant instruction removal
- parameter packing

### **Jump Chain Reduction**

This optimization looks for unconditional jumps to one or more Control Transfer Statement (CTS) instructions. The terminal CTS instruction is then replicated onto all of the non-terminal CTS instructions.

### **Useless Jump Removal**

This optimization looks for unconditional and conditional jumps to the next instruction. The jump is then removed.

### **Redundant Instruction Removal**

This optimization looks for redundant statements without side effects. One of the redundant statements is then removed.

### **Parameter Packing**

This optimization looks for adjacent parameter instructions and packs them into a more efficient parameter instruction.

## **Level 2: Local Optimizations**

Level 2 optimizations utilize a flowgraph to optimize the program one basic block at a time. Code motion, addition, and removal are limited to a basic block unit.

Level 2 optimizations include

- constant folding
- constant propagation
- global constant propagation
- definition removal
- string concatenation conversion
- parameter ordering

## **Constant Folding**

Expressions whose arguments are known at optimization time are evaluated. The result of the folded expression replaces the original expression with an assignment operation.

## **Constant Propagation**

The results of constant folding are tracked and propagated into other expressions using the value. These values can then be folded into other expressions.

## **Global Constant Propagation**

The values of symbols used in subsequent basic blocks are seeded for future Level 2 fold and propagate optimizations.

## **Definition Removal**

Expressions defining symbols that are later defined, without first being referenced, are removed. Expressions with side effects remove only the assignment to the symbol.

## **String Concatenation Conversion**

Chains of string concatenations are converted into a single and more efficient call to the PSL `join()` function.

## **Parameter Ordering**

Parameter statements with nested expressions are re-ordered so all expression evaluation occurs before the first parameter statement and all parameter statements immediately precede the call statement. This exposes Level 1 parameter packing opportunities.

## **Level 3: Global Optimizations**

Level 3 optimizations alter the flowgraph by adding, removing, and re-locating basic blocks.

Level 3 optimizations include

- block chain reduction
- unreachable code removal
- loop tail logic injection
- orphan block inlining

### **Block Chain Reduction**

One or more basic blocks that are entered from an unconditional jump are copied to the jump point. The resultant basic block is then extended to include the “copied” block.

### **Unreachable Code Removal**

Code that can never be executed is removed.

### **Loop Tail Logic Injection**

Loops with conditional heads and unconditional tails are converted so that the condition evaluation takes place in the loop’s tail(s). This decreases loop overhead by removing an unnecessary jump to the condition block.

### **Orphan Block Inlining**

Blocks that are entered exclusively via jumps are inlined (moved) into the innermost loop that references this block.

## **Optimization Criteria**

The PSL Optimizer decides at what level to optimize a program based on the following criteria:

- requested level
- demanded level
- maximum level

## Requested Level of Optimization

The requested level is implicitly defined based on the context of the call to the PSL Compiler. For example, a PSL parameter script will request Level 3 global optimization when it is compiled. A script submitted via the `%PSL` command will request the default level (Level 1 peephole) of optimization when it is compiled.

Presently, Level 3 global optimization is requested for all pre-discovery, discovery, and parameter PSL scripts. The default Level 1 is used for all other commands.

## Demanded Level of Optimization

The demanded level is explicitly defined and takes precedence over the requested level. The demanded level can be specified on the command line or directly in the PSL program via the pragma statement.

For example, a parameter can demand Level 2 optimization (and thus override the Level 3 request) by inserting the following pragma statement in the program:

```
pragma "O2";
```

---

### Note

---

Note that the O in the O2 in the line above is an uppercase alphabetic character and not a number zero.

---

Likewise, the optimization level can be specified system-wide via the command line:

```
PatrolAgent -02
```

---

### Note

---

Note that the 0 in the 02 in the line above is a numeric and not an alpha capital letter O.

---

This will cause all PSL scripts to be optimized at Level 2 except for any scripts containing a `pragma` statement requesting another level of optimization.

## Maximum Level of Optimization

The maximum level is explicitly defined and takes precedence over the demanded level and the requested level. The maximum level restricts the Optimizer tool from invoking higher levels. For example, if a parameter specifies a maximum optimization level of 1, it will never get optimized at Levels 2 or 3. A program specifying a maximum level can use a `pragma` statement like this:

```
pragma -OM1;
```

---

### Example

---

```
# This program will be optimized at level 1 and  
# print the levels 1 and 2 results for the  
# program and its processes.  
pragma "-01P2"
```

---

Likewise, the maximum level can be specified system-wide via the command line. For example,

```
PatrolAgent -OM1
```

Specifying a maximum level of zero (0) turns off the PSL Optimizer tool.

## Command-Line Specified Options

The PSL Optimizer tool can be controlled using the following methods:

- Via a command-line switch for the `psl` and `PatrolAgent` commands. This affects scripts system wide.
- Via the `pragma` PSL statement. This affects only the script using the `pragma` statement.

The format is as follows:

-O # M# P#

<b>Symbol</b>	<b>Description</b>
#	Specify the demanded optimization level. If no level is specified, the requested optimization level, defined implicitly based on the context of the PSL script, will be used. Valid values are 0, 1, 2 and 3.
M#	Specify the maximum optimization level. Valid values are 0, 1, 2 and 3. The default level is 3.
P#	Specify the level at which to print optimization results (to stdout). Valid values are: <ul style="list-style-type: none"><li>• 0 = Quiet, no results printed</li><li>• 1 = Optimization statistics results printed</li><li>• 2 = Execution results plus level 1 results</li><li>• 3 = Flow Graph program listing plus level 2 results</li></ul>



---

# Index

## Symbols

# character 3-2  
 %DUMP 4-2  
 %DUMP CHANNELS 4-3  
 %DUMP LIBRARIES 4-4  
 %PSL-execute PSL statement 4-6  
 %PSLPS-list PSL processes 4-7  
 {} 3-2

## A

ActiveX scripts 2-7  
 application class, built-in variables B-3  
 application discovery 1-9  
 application instance, built-in variables B-4  
 application instances 1-11  
 application objects 1-11  
 application variables 1-11  
 arithmetic operators 2-9  
 assignment operators 2-10

## B

backslash rules 2-6  
 bitwise operators 2-11

block chain reduction optimization C-14  
 built-in Agent namespace variables 1-13  
   for application class B-3  
   for application instance B-4  
   for computer B-2

## C

case, significance of 2-3  
 collector parameter 1-10  
 comments in PSL 3-2  
 common coding errors 5-8  
 compiler warnings 5-12  
 compound statements 3-2  
 computer class  
   built-in Agent namespace variables B-2  
 constant folding optimization C-13  
 constant propagation optimization C-13  
 consumer parameter 1-10

## D

data types, PSL 2-2  
 decrement operators 2-11  
 definition removal optimization C-13  
 diagnosing PSL errors 5-1

differences from C 5-7  
discovery script 1-9  
do...until statement 3-4

## E

efficiency 1-10  
errno return codes 5-6  
error checking, run-time 5-2  
error diagnosing 5-1  
errors  
    diagnosing 5-1  
exit statement 3-6  
exit\_status system return code 5-7  
export statement 3-7  
expressions, PSL 2-9  
external commands  
    %DUMP 4-2  
    %DUMP CHANNELS 4-3  
    %DUMP LIBRARIES 4-4  
    %PSL-execute PSL statement 4-6  
    %PSLPS-list PSL processes 4-7  
    psl-PSL compiler 4-8

## F

foreach statement 3-11  
function statement 3-13  
functions  
    backward compatibility 3-19  
    entry point 3-17  
    limitations of user-defined 3-19  
    local variables 3-16  
    return statement 3-14  
    start of execution 3-18

## G

global constant propagation optimization  
    C-13  
global optimizations C-13

## H

here document 2-6

## I

if statement 3-21  
incompatibilities with C 5-7  
increment/decrement operators 2-11  
intermediate code (quad) optimizer tool  
    C-10  
interpreted language 1-2

## J

jump chain reduction optimization C-12

## L

last statement 3-23  
Level 1 optimizations C-11  
Level 1 peephole optimization C-11  
Level 2 local optimizations C-12  
Level 3 global optimization C-13  
list values 2-8  
local optimizations C-12  
logical operators 2-11  
loop tail login injection optimization C-14  
lvalues 2-3

## N

- name space, common 2-3
- naming conventions, PSL 1-13
- new-line character 2-6
- next statement 3-24

## O

- operators, PSL 2-9
- optimization 1-2
- optimization levels C-11, C-12, C-13
- orphan block inlining optimization C-14

## P

- parameter built-in Agent namespace
  - variables B-5
- parameter ordering optimization C-13
- parameter packing optimization C-12
- PATROL Script Host, ActiveX 2-7
- peephole optimization C-11
- pitfalls, PSL 5-7
- ppv tool C-3, C-8
- ProfDefaultOptions() function C-5
- ProfGet() function C-6
- ProfGetTotalCpu() function C-6
- Profiler API C-3
- Profiler tool C-2
- Profiler Viewer tool C-3
  - profiling option C-4, C-8
- ProfOptions() function C-7
- ProfReset() function C-7
- ProfTop() function C-7
- PSL
  - common coding errors 5-8
  - compiler warnings 5-12
  - data types 2-2

- error diagnosing 5-1
- external commands 4-1
- here document 2-6
- incompatibilities with C 5-7
- operators 2-9
- pitfalls 5-7
- run-time error messages 5-15
- statements 3-2
- style guidelines 1-13
- tools

- additional C-1
- PSL Optimizer tool command line options
  - C-16
- PSL Profile Viewer (ppv) Tool C-8
- PSL Profiler API C-3
- PSL Profiler functions C-2
- PSL Profiler tool
  - description C-2
  - limitations C-4
  - with PATROL v3.1 C-3
- PSL Profiler Viewer (ppv) tool C-3
- PsIDebug variable 5-2
- psl-PSL compiler 4-8

## Q

- quad optimizer tool C-10

## R

- redundant instruction removal optimization
  - C-12
- relational operators 2-12
- relative address 1-12
- requires statement 3-25
- root directory 1-11
- run-time error checking 5-2
- run-time error messages 5-15

## S

- scalar 2-2
- scripts, ActiveX 2-7
- simple expressions 2-9
- statement block 3-2

### Statements

- do...until 3-4
- exit 3-6
- export 3-7
- foreach 3-11
- function 3-13
- if 3-21
- last 3-23
- next 3-24
- requires 3-25
- switch 3-28
- while 3-33

string concatenation conversion optimization C-13

- string literals 2-5
- string operators 2-13
- strings
  - here document 2-6
  - string literals 2-5
- switch statement 3-28

## T

- tab character 2-6

## U

- unreachable code removal optimization C-14
- useless jump removal optimization C-12
- user commands 1-10
- user-defined variables 1-13

## V

- variables
  - built-in 1-13, B-1
  - PsIDebug 5-2
  - user-defined 1-13

## W

- while statement 3-33

# 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.

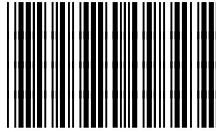
**LIMITATION OF LIABILITY.** Except as stated in the next succeeding paragraph, BMC's and your BMC reseller's total liability for all damages in connection with this License is limited to the price paid for the License. **IN NO EVENT SHALL BMC BE LIABLE FOR ANY CONSEQUENTIAL, SPECIAL, INCIDENTAL, PUNITIVE 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



\*17597\*