

# Performance Co-Pilot™ Programmer's Guide

007-3434-004

Version 2.2

---

## CONTRIBUTORS

Illustrated by Dany Galgani

Edited by Susan Wilkening

Production by Glen Traefald

Engineering and written contributions by David Chatterton, Michael Gigante, Mark Goodwin, Tony Kavadias, Seppo Keronen, Johnathon Knispel, Ken McDonell, Max Matveev, Ania Milewska, Daniel Moore, Heidi Muehlebach, Ivan Rayner, Nathan Scott, Timothy Shimmin, and Bill Tuthill

---

## COPYRIGHT

© 1996–1999, 2001 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

---

## LIMITED RIGHTS LEGEND

The electronic (software) version of this document was developed at private expense; if acquired under an agreement with the USA government or any contractor thereto, it is acquired as "commercial computer software" subject to the provisions of its applicable license agreement, as specified in (a) 48 CFR 12.212 of the FAR; or, if acquired for Department of Defense units, (b) 48 CFR 227-7202 of the DoD FAR Supplement; or sections succeeding thereto. Contractor/manufacture is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy 2E, Mountain View, CA 94043-1351.

---

## TRADEMARKS AND ATTRIBUTIONS

IRIX is a registered trademarks and Performance Co-Pilot, SGI, and the SGI logo are trademarks of Silicon Graphics, Inc. Cisco is a registered trademark of Cisco Systems, Inc. UNIX is a registered trademark in the United States and many other countries, licensed exclusively through X/Open Company Ltd.

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

---

## New Features in This Guide

Some extensions to the API have been documented for the Performance Co-Pilot release 2.2. New subsections include “Return Codes for `pmdaFetch` Callbacks,” Chapter 2; “`pmGetContextHostName` Function” and `pmGetConfig` Function,” Chapter 3; and the addition of counter metrics to the trace PDMA in Chapter 4.

In addition, miscellaneous formatting and editing changes were made throughout the document.



---

## Record of Revision

<b>Version</b>	<b>Description</b>
003	July 1999 Revised to support the Performance Co-Pilot release 2.1 for SGI systems running the IRIX 6.2, 6.3, 6.4, and 6.5 operating systems.
004	March 2001 Revised to support the Performance Co-Pilot release 2.2 for SGI systems running the IRIX 6.2, 6.3, 6.4, and 6.5.11 and later operating systems.



---

# Contents

<b>About This Guide</b>	<b>xxv</b>
What This Guide Contains	xxv
Audience for This Guide	xxv
Related Resources	xxvi
Man Pages	xxvi
Release Notes	xxvii
SGI Web Sites	xxvii
Obtaining Publications	xxvii
Conventions	xxviii
Reader Comments	xxix
<b>1. Programming Performance Co-Pilot</b>	<b>1</b>
PCP Architecture	1
Distributed Collection	3
Name Space	5
Distributed PMNS	5
Retrospective Sources of Performance Metrics	6
PMDA Development	6
Overview	7
Building a PMDA	7
In-Process (DSO) Method	7
Daemon Process Method	8
Shell Process Method	8
Client Development and PMAPI	8
Library Reentrancy and Threaded Applications	9
<b>007-3434-004</b>	<b>vii</b>

<b>2. Writing a PMDA</b>	<b>11</b>
Implementing a PMDA	11
PMDA Architecture	12
Overview	12
DSO PMDA	13
Daemon PMDA	15
Caching PMDA	16
Domains, Metrics, and Instances	16
Overview	16
Domains	18
Metrics	18
Data Structures	19
Semantics	21
Instances	22
N Dimensional Data	22
Data Structures	23
Other Issues	26
Extracting the Information	26
Latency and Threads of Control	27
Name Space	28
PMDA Help Text	29
Management of Evolution within a PMDA	30
DSO Interface	32
Overview	32
Trivial PMDA	33
Simple PMDA	33
simple_store in the Simple PMDA	36
Return Codes for pmdaFetch Callbacks	38
PMDA Structures	39



Initializing a PMDA . . . . .	41
Overview . . . . .	41
Common Initialization . . . . .	41
Trivial PMDA . . . . .	42
Simple PMDA . . . . .	42
Daemon Initialization . . . . .	43
Testing and Debugging a PMDA . . . . .	44
Overview . . . . .	45
Debugging Information . . . . .	45
dbpmda Debug Utility . . . . .	46
Integration of PMDA . . . . .	47
Installing a PMDA . . . . .	47
Upgrading a PMNS to Include Metrics from a New PMDA . . . . .	51
Removing a PMDA . . . . .	51
Configuring PCP Tools . . . . .	52
<b>3. PMAPI—The Performance Metrics API . . . . .</b>	<b>53</b>
Naming and Identifying Performance Metrics . . . . .	53
Performance Metric Instances . . . . .	54
Current PMAPI Context . . . . .	56
Performance Metric Descriptions . . . . .	57
Performance Metrics Values . . . . .	59
PMAPI Programming Style and Interaction . . . . .	62
Variable Length Argument and Results Lists . . . . .	63
PMAPI Error Handling . . . . .	63
PMAPI Procedural Interface . . . . .	64
PMAPI Name Space Services . . . . .	64
pmGetChildren Function . . . . .	64

pmGetChildrenStatus Function . . . . .	65
pmGetPMNSLocation Function . . . . .	65
pmLoadNameSpace Function . . . . .	66
pmLoadASCIINamespace Function . . . . .	66
pmLookupName Function . . . . .	67
pmNameAll Function . . . . .	67
pmNameID Function . . . . .	68
pmTraversePMNS Function . . . . .	68
pmTrimNameSpace Function . . . . .	68
pmUnloadNameSpace Function . . . . .	69
PMAPI Metrics Description Services . . . . .	69
pmLookupDesc Function . . . . .	69
pmLookupInDomText Function . . . . .	69
pmLookupText Function . . . . .	70
PMAPI Instance Domain Services . . . . .	70
pmGetInDom Function . . . . .	70
pmLookupInDom Function . . . . .	71
pmNameInDom Function . . . . .	71
PMAPI Context Services . . . . .	71
pmNewContext Function . . . . .	73
pmDestroyContext Function . . . . .	74
pmDupContext Function . . . . .	74
pmUseContext Function . . . . .	74
pmWhichContext Function . . . . .	74
pmAddProfile Function . . . . .	75
pmDelProfile Function . . . . .	75
pmSetMode Function . . . . .	75
pmReconnectContext Function . . . . .	78

---

pmGetContextHostName Function . . . . .	78
PMAPI Timezone Services . . . . .	79
pmNewContextZone Function . . . . .	79
pmNewZone Function . . . . .	79
pmUseZone Function . . . . .	80
pmWhichZone Function . . . . .	80
PMAPI Metrics Services . . . . .	80
pmFetch Function . . . . .	80
pmFreeResult Function . . . . .	82
pmStore Function . . . . .	82
PMAPI Record-Mode Services . . . . .	83
pmRecordAddHost Function . . . . .	83
pmRecordControl Function . . . . .	83
pmRecordSetup Function . . . . .	84
PMAPI Archive-Specific Services . . . . .	87
pmGetArchiveLabel Function . . . . .	87
pmGetArchiveEnd Function . . . . .	88
pmGetInDomArchive Function . . . . .	88
pmLookupInDomArchive Function . . . . .	89
pmNameInDomArchive Function . . . . .	89
pmFetchArchive Function . . . . .	89
PMAPI Time Control Services . . . . .	90
PMAPI Ancillary Support Services . . . . .	91
pmGetConfig Function . . . . .	91
pmErrStr Function . . . . .	92
pmExtractValue Function . . . . .	92
pmConvScale Function . . . . .	94
pmUnitsStr Function . . . . .	95

pmIDStr Function . . . . .	95
pmInDomStr Function . . . . .	95
pmTypeStr Function . . . . .	95
pmAtomStr Function . . . . .	96
pmNumberStr Function . . . . .	96
pmPrintValue Function . . . . .	96
pmFlush Function . . . . .	98
pmprintf Function . . . . .	98
pmSortInstances Function . . . . .	98
pmParseInterval Function . . . . .	99
pmParseMetricSpec Function . . . . .	99
PMAPI Programming Issues and Examples . . . . .	101
Symbolic Association between a Metric’s Name and Value . . . . .	101
Initializing New Metrics . . . . .	102
Iterative Processing of Values . . . . .	103
Accommodating Program Evolution . . . . .	103
Handling PMAPI Errors . . . . .	104
Compiling and Linking PMAPI Applications . . . . .	105
<b>4. Trace PMDA . . . . .</b>	<b>107</b>
Performance Instrumentation and Tracing . . . . .	107
Trace PMDA Design . . . . .	108
Application Interaction . . . . .	108
Sampling Techniques . . . . .	109
Simple Periodic Sampling . . . . .	109
Rolling-Window Periodic Sampling . . . . .	110
Rolling-Window Periodic Sampling Example . . . . .	111
Configuring the Trace PMDA . . . . .	113

Trace API . . . . .	114
Transactions . . . . .	115
Point Tracing . . . . .	116
Observations and Counters . . . . .	117
Configuring the Trace Library . . . . .	118
Instrumenting Applications to Export Performance Data . . . . .	120
<b>Appendix A. Acronyms . . . . .</b>	<b>123</b>
<b>Index . . . . .</b>	<b>125</b>



---

## Figures

<b>Figure 1-1</b>	PCP Global Process Architecture . . . . .	3
<b>Figure 1-2</b>	Process Structure for Distributed Operation . . . . .	4
<b>Figure 1-3</b>	Architecture for Retrospective Analysis . . . . .	6
<b>Figure 3-1</b>	A Structured Result for Performance Metrics from pmFetch . . . . .	60
<b>Figure 4-1</b>	Trace PMDA Overview . . . . .	108
<b>Figure 4-2</b>	Sample Duration Comparison . . . . .	111
<b>Figure 4-3</b>	Sampling Intervals . . . . .	112
<b>Figure 4-4</b>	Application and PCP Relationship . . . . .	120





---

## Tables

<b>Table 2-1</b>	Variables to Control Behavior of Generic <code>pmdaproc.sh</code> Procedures . . . . .	49
<b>Table 3-1</b>	Context Components of PMAPI Functions . . . . .	71
<b>Table 3-2</b>	Time Control Functions in PMAPI . . . . .	90
<b>Table 3-3</b>	PMAPI Type Conversion . . . . .	93
<b>Table 4-1</b>	Selected Command-Line Options . . . . .	113
<b>Table 4-2</b>	<code>trace.transact</code> Metrics . . . . .	115
<b>Table 4-3</b>	<code>trace.point</code> Metrics . . . . .	116
<b>Table 4-4</b>	<code>trace.observe</code> Metrics . . . . .	117
<b>Table 4-5</b>	Environment Variables . . . . .	118
<b>Table 4-6</b>	State Flags . . . . .	119
<b>Table A-1</b>	Performance Co-Pilot Acronyms and Their Meanings . . . . .	123



---

## Examples

<b>Example 2-1</b>	Simple PMDA as a DSO . . . . .	14
<b>Example 2-2</b>	Simple PMDA as a Daemon . . . . .	15
<b>Example 2-3</b>	<code>_pmID_int</code> Structure . . . . .	20
<b>Example 2-4</b>	<code>pmdaMetric</code> Structure . . . . .	20
<b>Example 2-5</b>	Trivial PMDA . . . . .	21
<b>Example 2-6</b>	Effect of Semantics on a Metric . . . . .	22
<b>Example 2-7</b>	<code>pmdaInstid</code> Structure . . . . .	23
<b>Example 2-8</b>	<code>pmdaIndom</code> Structure . . . . .	24
<b>Example 2-9</b>	<code>_pmInDom_int</code> Structure . . . . .	24
<b>Example 2-10</b>	Simple PMDA . . . . .	24
<b>Example 2-11</b>	<code>pmns</code> File for the Simple PMDA . . . . .	28
<b>Example 2-12</b>	Alternate <code>pmns</code> File for the Simple PMDA . . . . .	28
<b>Example 2-13</b>	Help Text for the Simple PMDA . . . . .	29
<b>Example 2-14</b>	Setting Values . . . . .	31
<b>Example 2-15</b>	Request Handling Callbacks in the Trivial PMDA . . . . .	33
<b>Example 2-16</b>	Request Handling Callbacks in the Simple PMDA . . . . .	33
<b>Example 2-17</b>	<code>simple.numfetch</code> Metric . . . . .	34
<b>Example 2-18</b>	<code>simple.color</code> Metric . . . . .	34
<b>Example 2-19</b>	<code>simple.time</code> Metric . . . . .	35
<b>Example 2-20</b>	<code>simple.now</code> Metric . . . . .	35
<b>Example 2-21</b>	<code>simple_store</code> in the Simple PMDA . . . . .	36
<b>Example 2-22</b>	<code>simple.color</code> and <code>PM_ERR_INST</code> Errors . . . . .	37
<b>Example 2-23</b>	<code>PM_ERR_P MID</code> Errors . . . . .	38

<b>Example 2-24</b>	-EACCES and PM_ERR_P MID Errors . . . . .	38
<b>Example 2-25</b>	pm daInterface Structure . . . . .	39
<b>Example 2-26</b>	pm daExt Structure . . . . .	40
<b>Example 2-27</b>	Initialization in the Trivial PMDA . . . . .	42
<b>Example 2-28</b>	Initialization in the Simple PMDA . . . . .	42
<b>Example 2-29</b>	main in the Simple PMDA . . . . .	44
<b>Example 2-30</b>	simple.numfetch in the Simple PMDA . . . . .	46
<b>Example 2-31</b>	Install Script for the Trivial PMDA . . . . .	48
<b>Example 2-32</b>	Changing Variable Assignments . . . . .	50
<b>Example 3-1</b>	Metrics Sharing the Same Instance Domain . . . . .	55
<b>Example 3-2</b>	pmDesc Structure . . . . .	57
<b>Example 3-3</b>	pmUnits and pmDesc Structures . . . . .	58
<b>Example 3-4</b>	pmValueBlock and pmValue Structures . . . . .	60
<b>Example 3-5</b>	pmValueBlock Structure . . . . .	61
<b>Example 3-6</b>	pmValueSet Structure . . . . .	61
<b>Example 3-7</b>	pmResult Structure . . . . .	62
<b>Example 3-8</b>	Dumping Values in Temporal Sequence . . . . .	77
<b>Example 3-9</b>	Replaying Interpolated Metrics . . . . .	77
<b>Example 3-10</b>	PM API Metrics Services . . . . .	81
<b>Example 3-11</b>	pmRecordHost Structure . . . . .	85
<b>Example 3-12</b>	pmLogLabel Structure . . . . .	87
<b>Example 3-13</b>	pmAtomValue Structure . . . . .	92
<b>Example 3-14</b>	Using pmPrintValue to Print Values . . . . .	97
<b>Example 3-15</b>	pmMetricSpec Structure . . . . .	100
<b>Example 3-16</b>	C Code Produced by pmgenmap Input . . . . .	102
<b>Example 3-17</b>	Initializing Metric Specifications . . . . .	102
<b>Example 3-18</b>	Iterative Processing . . . . .	103

<b>Example 3-19</b>	Adding a Metric . . . . .	103
<b>Example 3-20</b>	PMAPI Error Handling . . . . .	104
<b>Example 4-1</b>	Rolling-Window Sampling Technique . . . . .	109



---

## Procedures

<b>Procedure 2-1</b>	Creating a PMDA . . . . .	11
<b>Procedure 3-1</b>	Creating a Recording Session . . . . .	85





---

## About This Guide

This guide describes how to program Performance Co-Pilot (PCP), a software package of advanced performance management applications for the SGI family of graphical workstations and servers. PCP provides a systems-level suite of tools that cooperate to deliver distributed, integrated performance monitoring and performance management services spanning the hardware platform, the operating system, service layers, user applications, and distributed application architectures.

### What This Guide Contains

This guide contains the following chapters:

- Chapter 1, page 1, contains a thumbnail sketch of how to program the various PCP components.
- Chapter 2, page 11, describes how to write Performance Metrics Domain Agents (PMDAs) for PCP.
- Chapter 3, page 53, describes the interface that allows you to design custom performance monitoring tools.
- Chapter 4, page 107, provides an introduction to the design of the trace Performance Metrics Domain Agent (PMDA).
- Appendix A, page 123, provides a comprehensive list of the acronyms used in this guide, in the PCP man pages, and in the release notes.

### Audience for This Guide

The guide describes the programming interfaces to Performance Co-Pilot (PCP) for the following intended audience:

- Performance analysts or system administrators who want to extend or customize performance monitoring tools available with PCP
- Developers who need to integrate their applications into the PCP framework

This book is written for those who are competent with the C programming language, the UNIX operating system, and the target domain from which the desired performance metrics are to be extracted. Familiarity with the PCP tool suite is assumed.

## Related Resources

The *Performance Co-Pilot User's and Administrator's Guide*, which is a companion document to the *Performance Co-Pilot Programmer's Guide*, is intended for system administrators and performance analysts who are directly using and administering PCP applications.

Additional resources include man pages, release notes, and SGI Web sites.

## Man Pages

The IRIX man pages provide concise reference information on the use of IRIX commands, subroutines, and system resources. There is usually a man page for each PCP command or subroutine. To see a list of all the PCP man pages, enter the following command:

```
man -k performance
```

To see a particular man page, supply its name to the man command, for example:

```
man pcp
```

The man pages are divided into the following seven sections:

- (1) General commands
- (2) System calls and error numbers
- (3) Library subroutines
- (4) File formats
- (5) Miscellaneous

- (6) Demos and games
- (7) Special files

When referring to man pages, this guide follows a standard UNIX convention: the section number in parentheses follows the item. For example, `pmda(3)` refers to the man page in section 3 for the `pmda` command.

## Release Notes

Release notes provide specific information about the current product release, available online through the `relnotes` command. Exceptions to the printed and online documentation are found in the release notes. The `grelnotes` command provides a graphical interface to the release notes of all products installed on your system. For additional information, see the `relnotes(1)` and `grelnotes(1)` man pages.

## SGI Web Sites

The following Web sites are accessible to everyone with general Internet access:

URL	Description
<code>http://www.sgi.com</code>	The SGI general Web site, with search capability
<code>http://www.sgi.com/software</code>	Links to Performance Co-Pilot product information
<code>http://oss.sgi.com/projects/pcp</code>	Some parts of the PCP infrastructure that have also been released as open source

## Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at:

`http://techpubs.sgi.com`.

## Conventions

The following conventions are used throughout this document:

Convention	Meaning
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.
<b>user input</b>	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
ALL CAPS	All capital letters denote environment variables, operator names, directives, defined constants, and macros in C programs.
()	Parentheses that follow function names surround function arguments or are empty if the function has no arguments; parentheses that follow IRIX commands surround man page section numbers.

## Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

`techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

`http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

Technical Publications  
SGI  
1600 Amphitheatre Pkwy., M/S 535  
Mountain View, California 94043-1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

We value your comments and will respond to them promptly.



## Programming Performance Co-Pilot

Performance Co-Pilot (PCP) provides a systems-level suite of tools that cooperate to deliver distributed, integrated performance management services. PCP is designed for the in-depth analysis and sophisticated control that are needed to understand and manage the hardest performance problems in our most complex systems.

PCP provides unparalleled power to quickly isolate and understand performance behavior, resource utilization, activity levels and performance bottlenecks.

Performance data may be collected and exported from multiple sources, most notably the hardware platform, the IRIX kernel, layered services, and end-user applications.

There are several ways to extend PCP by programming certain of its components:

- By writing a Performance Metrics Domain Agent (PMDA) to collect performance metrics from an uncharted performance domain (Chapter 2, page 11)
- By creating new analysis or visualization tools using documented functions from the Performance Metrics Application Programming Interface (PMAPI) (Chapter 3, page 53)
- By adding performance instrumentation to an application using the trace facilities of the PCP trace library (`libpcp_trace`) and the trace PMDA (Chapter 4, page 107)

In addition, the topic of customizing a PCP installation is covered in the chapter on customizing and extending PCP service in the *Performance Co-Pilot User's and Administrator's Guide*.

### 1.1 PCP Architecture

This section gives a brief overview of PCP architecture. For an explanation of terms and acronyms, refer to Appendix A, page 123.

PCP consists of several monitoring and collecting tools. Monitoring tools such as `pmchart` and `pmview` visualize metrics, but have minimal interaction with target systems; see the `pmchart(1)` and `pmview(1)` man pages. Collection tools, called PMDAs, extract performance values from target systems, but do not provide graphical user interfaces.

Systems supporting PCP services are broadly classified into two categories:

Collector	Hosts that have the PMCD and one or more PMDAs running to collect and export performance metrics
Monitor	Hosts that import performance metrics from one or more collector hosts to be consumed by tools to monitor, manage, or record the performance of the collector hosts

Each PCP enabled host can operate as a collector, or a monitor, or both.

There are separate node-locked licenses for collector and monitor functions.

Figure 1-1 shows the architecture of PCP. The monitoring tools consume and process performance data using a public interface, the Performance Metrics Application Programming Interface (PMAPI).

Below the PMAPI level is the PMCD process, which acts in a coordinating role, accepting requests from clients, routing requests to one or more PMDAs, aggregating responses from the PMDAs, and responding to the requesting client.

Each performance metric domain (such as IRIX or some Database Management System (DBMS)) has a well-defined name space for referring to the specific performance metrics it knows how to collect.



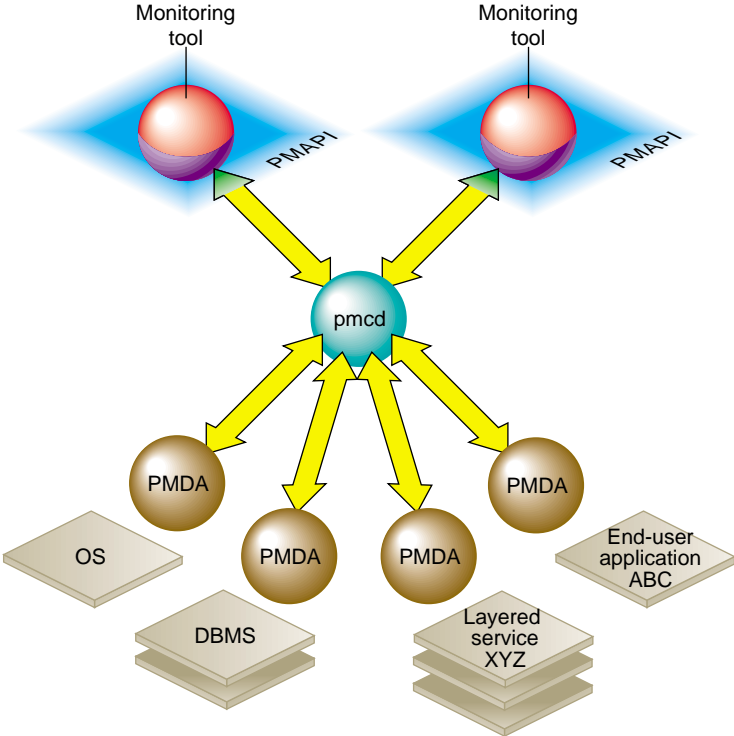
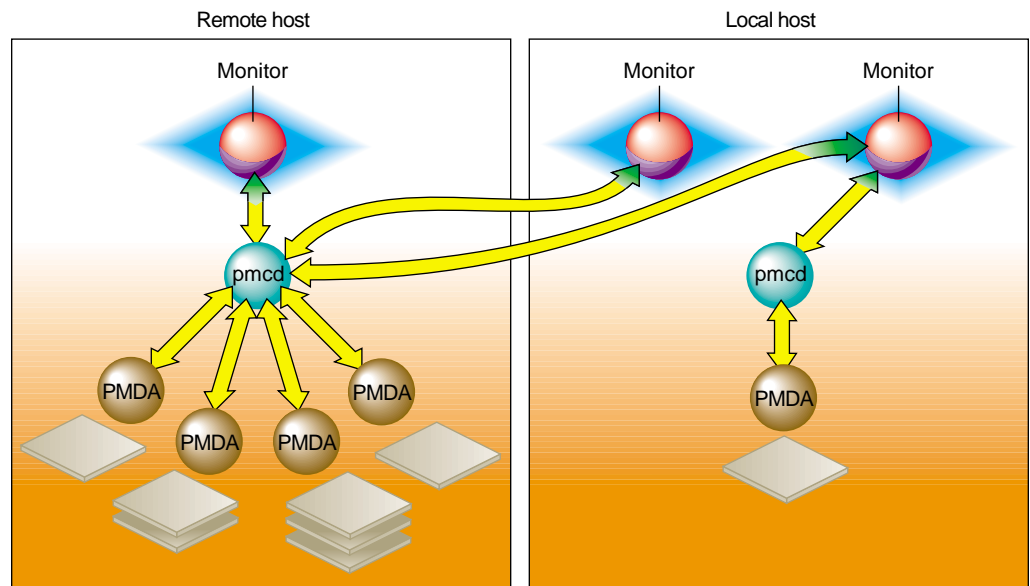


Figure 1-1 PCP Global Process Architecture

### 1.1.1 Distributed Collection

The performance metrics collection architecture is distributed, in the sense that any monitoring tool may be executing remotely. However, a PMDA is expected to be running on the system for which it is collecting performance measurements; there are some notable PMDAs such as Cisco and Array that are exceptions, and collect performance data from remote systems.

As shown in Figure 1-2, monitoring tools communicate only with PMCD. The PMDAs are controlled by PMCD and respond to requests from the monitoring tools that are forwarded by PMCD to the relevant PMDAs on the collector host.



**Figure 1-2** Process Structure for Distributed Operation

The host running the monitoring tools does not require any collection tools, including PMCD, since all requests for metrics are sent to the PMCD process on the collector host.

The connections between monitoring tools and PMCD processes are managed in `libpcp`, below the PMAPI level; see the `pmapi(3)` man page. Connections between PMDAs and PMCD are managed by the PMDA functions; see the `pmda(3)` and `pmcd(1)` man pages. There can be multiple monitor clients and multiple PMDAs on the one host, but there may be only one PMCD process.

### 1.1.2 Name Space

Each PMDA provides a domain of metrics, whether they be for the IRIX system, a database manager, a layered service, or an application module. These metrics are referred to by name inside the user interface, and with a numeric Performance Metric Identifier (PMID) within the underlying PMAPI.

The PMID consists of three fields: the domain, the cluster, and the item number of the metric. The domain is a unique number assigned to each PMDA. For example, two metrics with the same domain number must be from the same PMDA. The cluster and item numbers allow metrics to be easily organized into groups within the PMDA, and provide a hierarchical taxonomy to guarantee uniqueness within each PMDA.

The Performance Metrics Name Space (PMNS) describes the exported performance metrics, in particular the mapping from PMID to external name, and vice-versa.

### 1.1.3 Distributed PMNS

In PCP 1.x releases, the PMNS was required to be local to the application that referred to PCP metrics by name. As of release 2.0, PMNS operations by default are directed to the host or archive that is the source of the desired performance metrics.

In Figure 1-2, both Performance Metrics Collection Daemon (PMCD) processes would respond to PMNS queries from monitoring tools by referring to their local PMNS. If different PMDAs were installed on the two hosts, then the PMNS used by each PMCD would be different, to reflect variations in available metrics on the two hosts.

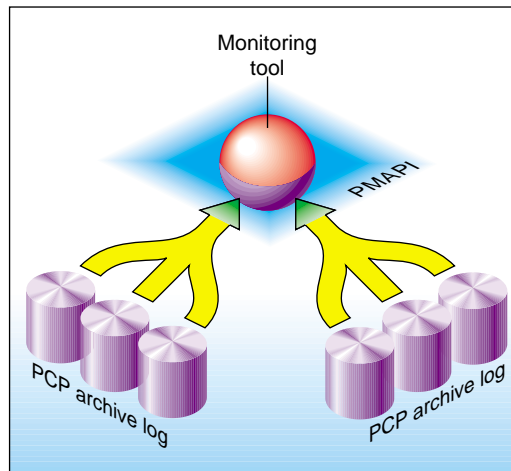
Distributed PMNS services necessitated changes to PCP protocols between client applications and PMCD, and to the internal format of PCP archive files. Release 2.x is compatible with earlier releases, so new PCP components operate correctly with either new or old PCP components. For example, when a PCP 2.x monitoring tool connects to PCP 1.x PMCD, or attempts to process a PCP archive created by PCP 1.x `pmlogger`, the monitoring tool reverts to using the local PMNS.

The `-n pmnsfile` option may be used with all PCP monitoring tools to force use of the local PMNS in preference to the PMNS at the source of the metrics.

### 1.1.4 Retrospective Sources of Performance Metrics

The distributed collection architecture described in the previous section is used when PMAPI clients are requesting performance metrics from a real-time or live source.

The PMAPI also supports delivery of performance metrics from a historical source in the form of a PCP archive log. Archive logs are created using the `pmlogger` utility, and are replayed in an architecture as shown in Figure 1-3.



**Figure 1-3** Architecture for Retrospective Analysis

## 1.2 PMDA Development

A collection of Performance Metrics Domain Agents (PMDAs) are provided with PCP to extract performance metrics. Each PMDA encapsulates domain-specific knowledge and methods about performance metrics that implement the uniform access protocols and functional semantics of the PCP. There is one PMDA for the operating system, another for process specific statistics, one each for common DBMS products, and so on. Thus, the range of performance metrics can be easily extended by implementing and integrating new PMDAs. Chapter 2, page 11, is a step-by-step guide to writing your own PMDA.

## 1.2.1 Overview

Once you are familiar with the PCP and PMDA frameworks, you can quickly implement a new PMDA with only a few data structures and functions. This book contains detailed discussions of PMDA architecture and the integration of PMDAs into the PCP framework. This includes integration with PMCD. However, details of extracting performance metrics from the underlying instrumentation vary from one domain to another and are not covered in this book.

A PMDA is responsible for a set of performance metrics, in the sense that it must respond to requests from PMCD for information about performance metrics, instance domains, and instantiated values. The PMCD process generates requests on behalf of monitoring tools that make requests using PMAPI functions.

You can incorporate new performance metrics into the PCP framework by creating a PMDA, then reconfiguring PMCD to communicate with the new PMDA.

## 1.2.2 Building a PMDA

A PMDA interacts with PMCD across one of several well-defined interfaces and protocol mechanisms. These implementation options are described in the *Performance Co-Pilot User's and Administrator's Guide*.

It is strongly recommended that code for a new PMDA be based on the source of one of the demonstration PMDAs below the `/var/pcp/pmdas` directory.

### 1.2.2.1 In-Process (DSO) Method

This method of building a PMDA uses a Dynamic Shared Object (DSO) that is attached by PMCD, using `dlopen`, at initialization time. This is the highest performance option (there is no context switching and no interprocess communication (IPC) between the PMCD and the PMDA), but is operationally intractable in some situations. For example, difficulties arise where special access permissions are required to read the instrumentation behind the performance metrics, or where the performance metrics are provided by an existing process with a different protocol interface. The DSO PMDA effectively executes as part of PMCD; so care is required when crafting a PMDA in this manner.

Also, multiple object code formats for the DSO may be required because PMCD must execute with the same object code format as the running IRIX kernel. This would be `o32` for some low-end platforms (IRIX 6.3 and earlier), `n32` for other low-end platforms (IRIX 6.5 and later), and `n64` for high-end platforms.

### 1.2.2.2 Daemon Process Method

Functionally, this method may be thought of as a DSO implementation with a standard `main` routine conversion wrapper so that communication with PMCD uses message passing rather than direct procedure calls. (See `/var/pcp/pmdas/trivial/trivial.c` file.)

The daemon PMDA is actually the most common, because it allows multiple threads of control, permits linking with existing dynamic libraries, and provides more resilient error encapsulation than the DSO method.

### 1.2.2.3 Shell Process Method

This method offers the least performance, but may be well-suited for rapid prototyping of performance metrics, or for diagnostic metrics that are not going into production.

Implementation of the ASCII protocols is rather lengthy. The suggested approach is to take the `/var/pcp/pmdas/news/pmdanews` PMDA as an illustrative example, and adapt it for the particular metrics of interest.

---

**Note:** The ASCII protocols have not been extensively used; so their support may be discontinued in a future PCP release. Newer versions of the PMDA libraries have dramatically reduced the code development effort required for a new PMDA (either the DSO or daemon approach), thereby reducing the need for ASCII protocols.

---

## 1.3 Client Development and PMAPI

Application developers are encouraged to create new PCP client applications to monitor, display, and analyze performance data in a manner suited to their particular site, application suite, or information processing environment.

PCP client applications are programmed using the Performance Metrics Application Programming Interface (PMAPI), documented in Chapter 3, page 53. The PMAPI, which provides performance tool developers with access to all of the distributed services of the Performance Metrics Collection Subsystem (PMCS), is the interface used by the standard PCP utilities.

Source for a sample PMAPI client may be found in the `/var/pcp/demos/pmclient` directory if the `pcp.sw.demo` subsystem has been installed.

## 1.4 Library Reentrancy and Threaded Applications

Most of the PCP libraries are not thread safe. This is a deliberate design decision to trade-off commonly required performance and efficiency against the less common requirement for multiple threads of control to call the PCP libraries.

The simplest and safest programming model is to designate at most one thread to make calls into the PCP libraries. This approach applies to both PMDAs using `libpcp_pmda` and monitoring applications using PMAPI and calling the `libpcp` library.

An important exception is the `libpcp_trace` library for instrumenting applications; it is thread safe.

Particular care should be taken with the utility functions in the `libpcp` library; for example, `pmprintf` and `pmflush` share a buffer that may be corrupted if calls to these functions from multiple threads are overlapped.





## Writing a PMDA

This chapter constitutes a programmer's guide to writing a Performance Metrics Domain Agent (PMDA) for Performance Co-Pilot (PCP).

The presentation assumes the developer is using the standard PCP `libpcp_pmda` library, as documented in the `pmda(3)` and associated man pages.

### 2.1 Implementing a PMDA

The job of a PMDA is to gather performance data and report them to the Performance Metrics Collection Daemon (PMCD) in response to requests from PCP monitoring tools routed to the PMDA via PMCD.

An important requirement for any PMDA is that it have low latency response to requests from PMCD. Either the PMDA must use a quick access method and a single thread of control, or it must have asynchronous refresh and two threads of control: one for communicating with PMCD, the other for updating the performance data.

The PMDA is typically acting as a gateway between the target domain (that is, the performance instrumentation in an application program or service) and the PCP framework. The PMDA may extract the information using one of a number of possible export options that include a shared memory segment or `mmap` file; a sequential log file (where the PMDA parses the tail of the log file to extract the information); a snapshot file (the PMDA rereads the file as required); or application-specific communication services (IPC). The choice of export methodology is typically determined by the source of the instrumentation (the target domain) rather than by the PMDA.

Procedure 2-1 describes the suggested steps for designing and implementing a PMDA:

**Procedure 2-1** Creating a PMDA

1. Determine how to extract the metrics from the target domain.
2. Select an appropriate architecture for the PMDA (daemon or DSO, IPC, `sproc`).
3. Define the metrics and instances that the PMDA will support.
4. Implement the functionality to extract the metric values.

5. Assign Performance Metric Identifiers (PMIDs) for the metrics, along with names for the metrics in the Performance Metrics Name Space (PMNS).
6. Specify the help file and control data structures for metrics and instances that are required by the standard PMDA implementation library functions.
7. Write code to supply the metrics and associated information to PMCD.
8. Implement any PMDA-specific callbacks, and PMDA initialization functions.
9. Exercise and test the PMDA with the purpose-built PMDA debugger; see the `dbpmda(1)` man page.
10. Install and connect the PMDA to a running PMCD process; see the `pmcd(1)` man page.
11. Configure or develop tools to use the new metrics. For examples of visualization tools, see the `pmchart(1)`, `pmgadgets(1)`, and `pmview(1)` man pages. For examples of alarm tools, see the `pmie(1)` and `pmieconf(1)` man pages.

Where appropriate, define `pmlogger` configurations suitable for creating PCP archives containing the new metrics. For more information, see the `pmlogger(1)` man page.

## 2.2 PMDA Architecture

This section discusses the two methods of connecting a PMDA to a PMCD process:

- As a separate process using some interprocess communication (IPC) protocol.
- As a dynamically attached library (that is, a dynamic shared object or DSO (for more details, see the `DSO(5)` man page).

### 2.2.1 Overview

All PMDAs are launched and controlled by the PMCD process on the local host. PMCD receives requests from the monitoring tools and forwards them to the PMDAs. Responses, when required, are returned through PMCD to the clients. The requests fall into a small number of categories, and the PMDA must handle each request type. For a DSO PMDA, each request type corresponds to a method in the agent. For a daemon PMDA, each request translates to a message or protocol data unit (PDU) that may be sent to a PMDA from PMCD.

For daemon PMDA, the following request PDUs must be supported:

PDU_FETCH	Request for metric values (see the <code>pmFetch(3)</code> man page.)
PDU_PROFILE	A list of instances required for the corresponding metrics in subsequent fetches (see the <code>pmAddProfile(3)</code> man page).
PDU_INSTANCE_REQ	Request for a particular instance domain for instance descriptions (see the <code>pmGetInDom(3)</code> man page).
PDU_DESC_REQ	Request for metadata describing metrics (see the <code>pmLookupDesc(3)</code> man page).
PDU_TEXT_REQ	Request for metric help text (see the <code>pmLookupText(3)</code> man page).
PDU_RESULT	Values to store into metrics (see the <code>pmStore(3)</code> man page).

Each PMDA is associated with a unique domain number that is encoded in the domain field of metric and instance identifiers, and PMCD uses the domain number to determine which PMDA can handle the components of any given client request.

## 2.2.2 DSO PMDA

Each PMDA is required to implement a function that handles each of the request types. By implementing these functions as library functions, a PMDA can be implemented as a dynamically shared object (DSO) and attached by PMCD at run time with the `dlopen` call; see the `dlopen(3)` man page. This eliminates the need for an IPC layer (typically a UNIX pipe) between each PMDA and PMCD, because each request becomes a function call rather than a message exchange. The required library functions are detailed in Section 2.5, page 32.

A PMDA that interacts with PMCD in this fashion must abide by a formal initialization protocol so that PMCD can discover the location of the library functions that are subsequently called with function pointers. When a DSO PMDA is installed, the PMCD configuration file, `/etc/pmcd.conf`, is updated to reflect the domain and name of the PMDA, the location of the shared object, and the name of the initialization function. The initialization sequence is discussed in Section 2.6, page 41.

As superuser, install the simple PMDA as a DSO, as shown in Example 2-1, and observe the changes in the PMCD configuration file. The output may differ slightly depending on the other PMDAs you have installed.

**Example 2-1** Simple PMDA as a DSO

```
# cd /var/pcp/pmdas/simple
# cat /etc/pmcd.conf
# Name Id IPC IPC Params File/Cmd
irix 1 dso irix_init libirixpmda.so
pmcd 2 dso pmcd_init pmda_pmcd.so
proc 3 dso proc_init pmda_proc.so
# ./Install
You will need to choose an appropriate configuration for installation
of the ``simple`` Performance Metrics Domain Agent (PMDA).
collector collect performance statistics on this system
monitor allow this system to monitor local and/or remote systems
both collector and monitor configuration for this system
Please enter c(ollector) or m(onitor) or b(oth) [b] both

Updating the Performance Metrics Name Space (PMNS) ...
Installing pmchart view(s) ...
Install simple as a daemon or dso agent? [daemon] dso
...
Check simple metrics have appeared ... 5 metrics and 9 values
# cat /etc/pmcd.conf
# Name Id IPC IPC Params File/Cmd
irix 1 dso irix_init libirixpmda.so
pmcd 2 dso pmcd_init pmda_pmcd.so
proc 3 dso proc_init pmda_proc.so
simple 253 dso simple_init pmda_simple.so
```

As can be seen from the contents of `/etc/pmcd.conf`, the DSO version of the simple PMDA is in a library named `pmda_simple.so` and has an initialization function called `simple_init`. The domain of the simple PMDA is 253, as shown in the column headed `Id`.

### 2.2.3 Daemon PMDA

A DSO PMDA provides the most efficient communication between the PMDA and PMCD. This approach has some disadvantages resulting from the DSO PMDA being the same process as PMCD:

- An error or bug that causes a DSO PMDA to exit also causes PMCD to exit.
- There is only one thread of control in PMCD; as a result, a computationally expensive PMDA, or worse, a PMDA that blocks for I/O, adversely affects the performance of PMCD.
- The PMCD runs as superuser; so any DSO PMDA's also run as superuser.
- A memory leak in a DSO PMDA also causes a memory leak for PMCD.

Consequently, many PMDA's are implemented as a daemon process.

The `libpcp_pmda` library is designed to allow simple implementation of a PMDA that runs as a separate process. The library functions provide a message passing layer acting as a generic wrapper that accepts PDUs, makes library calls using the standard DSO PMDA interface, and sends PDUs. Therefore, you can implement a PMDA as a DSO and then install it as either a daemon or a DSO, depending on the presence or absence of the generic wrapper.

The PMCD process launches a daemon PMDA with `fork` and `execv`. You can easily connect a pipe to the PMDA using standard input and output. The PMCD process may also connect to a daemon PMDA using TCP/IP or UNIX domain sockets; see the `inet(7)` or `unix(7)` man page.

As superuser, install the simple PMDA as a daemon process as shown in Example 2-2. As in Example 2-1, the output may differ due to other PMDA's already installed.

**Example 2-2** Simple PMDA as a Daemon

The specification for the simple PMDA now states the connection type of pipe to PMCD and the executable image for the PMDA is `/var/pcp/pmdas/simple/pmdasimple`, using domain number 253.

```
# cd /var/pcp/pmdas/simple
# ./Install
...
Install simple as a daemon or dso agent? [daemon] daemon
PMCD should communicate with the daemon via pipe or socket? [pipe] pipe
...
```

```
# cat /etc/pmcd.conf
# Name  Id   IPC      IPC Params  File/Cmd
irix    1    dso      irix_init   libirixpmda.so
pmcd    2    dso      pmcd_init   pmda_pmcd.so
proc    3    dso      proc_init   pmda_proc.so
simple   253  pipe     binary      /var/pcp/pmdas/simple/pmdasimple -d 253
```

## 2.2.4 Caching PMDA

When either the cost or latency associated with collecting performance metrics is high, the PMDA implementer may choose to trade off the currency of the performance data to reduce the PMDA resource demands or the fetch latency time.

One scheme for doing this is called a caching PMDA, which periodically instantiates values for the performance metrics and responds to each request from PMCD with the most recently instantiated (or cached) values, as opposed to instantiating current values on demand when the PMCD asks for them.

The Cisco PMDA is an example of a caching PMDA. For additional information, see the contents of the `/var/pcp/pmdas/cisco` directory and the `pmdacisco(1)` man page.

## 2.3 Domains, Metrics, and Instances

This section defines metrics and instances, discusses how they should be designed for a particular target domain, and shows how to implement support for them.

The examples in this section are drawn from the trivial and simple PMDAs, which are distributed in source format with PCP. Refer to the `/var/pcp/pmdas/trivial` and `/var/pcp/pmdas/simple` directories, respectively.

### 2.3.1 Overview

*Domains* are autonomous performance areas, such as the operating system or a layered service or a particular application. *Metrics* are raw performance data for a domain, and typically quantify activity levels, resource utilization or quality of service. *Instances* are sets of related metrics, as for multiple processors, or multiple service classes, or multiple transaction types.

PCP employs the following simple and uniform data model to accommodate the demands of performance metrics drawn from multiple domains:

- Each metric has an identifier that is unique across all metrics for all PMDAs on a particular host.
- Externally, metrics are assigned names for user convenience—typically there is a 1:1 relationship between a metric name and a metric identifier.
- The PMDA implementation determines if a particular metric has a singular value or a set of (zero or more) values. For instance, the metric `hinv.ndisk` counts the number of disks and has only one value on a host, whereas the metric `disk.dev.total` counts disk I/O operations and has one value for each disk on the host.
- If a metric has a set of values, then members of the set are differentiated by instances. The set of instances associated with a metric is an *instance domain*. For example, the set of metrics `disk.dev.total` is defined over an instance domain that has one member per disk spindle.

The selection of metrics and instances is an important design decision for a PMDA implementer. The metrics and instances for a target domain should have the following qualities:

- Obvious to a user
- Consistent across the domain
- Accurately representative of the operational and functional aspects of the domain

For each metric, you should also consider these questions:

- How useful is this value?
- What units give a good sense of scale?
- What name gives a good description of the metric's meaning?
- Can this metric be combined with another to convey the same useful information?

As with all programming tasks, expect to refine the choice of metrics and instances several times during the development of the PMDA.

### 2.3.2 Domains

Each PMDA must be uniquely identified by PMCD so that requests from clients can be efficiently routed to the appropriate PMDA. The unique identifier, the PMDA's domain, is encoded within the metrics and instance domain identifiers so that they are associated with the correct PMDA, and so that they are unique, regardless of the number of PMDAs that are connected to the PMCD process.

The default domain number for each PMDA is defined in `/var/pcp/pmns/stdpmid`. This file is a simple table of PMDA names and their corresponding domain number. However, a PMDA does not have to use this domain number—the file is only a guide to help avoid domain number clashes when PMDAs are installed and activated.

The domain number a PMDA uses is passed to the PMDA by PMCD when the PMDA is launched. Therefore, any data structures that require the PMDA's domain number must be set up when the PMDA is initialized, rather than declared statically. The protocol for PMDA initialization provides a standard way for a PMDA to implement this run-time initialization.

---

**Tip:** Although uniqueness of the domain number in the `/etc/pmcd.conf` control file used by PMCD is all that is required for successful starting of PMCD and the associated PMDAs, the developer of a new PMDA is encouraged to add the default domain number for each new PMDA to the `/var/pcp/pmns/stdpmid.local` file and then to run the `Make.stdpmid` script in `/var/pcp/pmns` to recreate `/var/pcp/pmns/stdpmid`; this file acts as a repository for documenting the known default domain numbers.

---

### 2.3.3 Metrics

A PMDA provides support for a collection of metrics. In addition to the obvious performance metrics, and the measures of time, activity and resource utilization, the metrics should also describe how the target domain has been configured, as this can greatly affect the correct interpretation of the observed performance. For example, metrics that describe network transfer rates should also describe the number and type of network interfaces connected to the host.



In addition, the metrics should describe how the PMDA has been configured. For example, if the PMDA was periodically probing a system to measure quality of service, there should be metrics for the delay between probes, the number of probes attempted, plus probe success and failure counters. It may also be appropriate to allow values to be stored (see the `pmstore(1)` man page) into the delay metric, so that the delay used by the PMDA can be altered dynamically.

### 2.3.3.1 Data Structures

Each metric must be described in a `pmDesc` structure; see the `pmLookupDesc(3)` man page:

```
typedef struct {
    pmID      pmid;          /* unique identifier */
    int       type;         /* base data type */
    pmInDom   indom;       /* instance domain */
    int       sem;         /* semantics of value */
    pmUnits   units;       /* dimension and units */
} pmDesc;
```

This structure contains the following fields:

<code>pmid</code>	A unique identifier, Performance Metric Identifier (PMID), that differentiates this metric from other metrics across the union of all PMDAs
<code>type</code>	A data type indicator showing whether the format is an integer (32 or 64 bit, signed or unsigned); float; double; string; or arbitrary aggregate of binary data
<code>indom</code>	An instance domain identifier that links this metric to an instance domain
<code>sem</code>	An encoding of the value's semantics (counter, instantaneous, or discrete)
<code>units</code>	A description of the value's units based on dimension and scale in the three orthogonal dimensions of space, time, and count (or events)

Symbolic constants of the form `PM_TYPE_*`, `PM_SEM_*`, `PM_SPACE_*`, `PM_TIME_*`, and `PM_COUNT_*` are defined in the `/usr/include/pcp/pmapi.h` file. You may use them to initialize the elements of a `pmDesc` structure. The `pmID` type is an unsigned integer that can be safely cast to a `_pmID_int` structure, which contains fields defining the metric's (PMDA's) domain, cluster, and item number as shown in Example 2-3:

**Example 2-3** `_pmID_int` Structure

```
typedef struct {
    int          pad:2;
    unsigned int domain:8;
    unsigned int cluster:12;
    unsigned int item:10;
} _pmID_int;
```

For additional information, see the `/usr/include/pcp/impl.h` file.

The `pad` field should be ignored. The domain number should be set at run time when the PMDA is initialized. The `PMDA_PMIID` macro defined in `/usr/include/pcp/pmapi.h` can be used to set the `cluster` and `item` fields at compile time, as these should always be known and fixed for a particular metric.

---

**Note:** The three components of the PMID should correspond exactly to the three-part definition of the PMID for the corresponding metric in the PMNS described in Section 2.4.3, page 28.

---

A table of `pmdaMetric` structures should be defined within the PMDA, with one structure per metric as shown in Example 2-4.

**Example 2-4** `pmdaMetric` Structure

```
typedef struct {
    void      *m_user;          /* for users external use */
    pmDesc    m_desc;          /* metric description */
} pmdaMetric;
```

This structure contains a `pmDesc` structure and a handle that allows PMDA-specific structures to be associated with each metric. For example, `m_user` could be a pointer to a global variable containing the metric value, or a pointer to a function that may be called to instantiate the metric's value.

The trivial PMDA, shown in Example 2-5, has only a singular metric (that is, no instance domains):

**Example 2-5** Trivial PMDA

```
static pmdaMetric metrictab[] = {
/* time */
  { (void *)0,
    { PMDA_PMID(0,1), PM_TYPE_U32, PM_INDOM_NULL, PM_SEM_INSTANT,
      PMDA_PMUNITS (0, 1, 0, 0, PM_TIME_SEC, 0) }, )
};
```

This single metric (`trivial.time`) has the following:

- A PMID with a cluster of 0 and an item of 1
- An unsigned 32-bit integer (`PM_TYPE_U32`)
- A singular value and hence no instance domain (`PM_INDOM_NULL`)
- An instantaneous semantic value (`PM_SEM_INSTANT`)
- Dimension “time” and the units “seconds”

### 2.3.3.2 Semantics

The metric’s semantics describe how PCP tools should interpret the metric’s value. The following are the possible semantic types:

- Counter (`PM_SEM_COUNTER`)
- Instantaneous value (`PM_SEM_INSTANT`)
- Discrete value (`PM_SEM_DISCRETE`)

A counter should be a value that monotonically increases (or monotonically decreases, which is less likely) with respect to time, so that the rate of change should be used in preference to the actual value. Rate conversion is not appropriate for metrics with instantaneous values, as the value is a snapshot and there is no basis for assuming any values that might have been observed between snapshots. Discrete is similar to instantaneous; however, once observed it is presumed the value will persist for an extended period (for example, system configuration, static tuning parameters and most metrics with nonnumeric values).

For a given time interval covering six consecutive timestamps, each spanning two units of time, the metric values, in Example 2-6, are exported from a PMDA (“N/A” implies no value is available):

**Example 2-6** Effect of Semantics on a Metric

Timestamps:	1	3	5	7	9	11
Value:	10	30	60	80	90	N/A

The default display of the values would be as follows:

Timestamps:	1	3	5	7	9	11
Semantics:						
Counter	N/A	10	15	10	5	N/A
Instantaneous	10	30	60	80	90	N/A
Discrete	10	30	60	80	90	90

### 2.3.4 Instances

Singular metrics have only one value and no associated instance domain. Some metrics contain a set of values that share a common set of semantics for a specific instance, such as one value per processor, or one value per disk spindle, and so on.

---

**Note:** The PMDA implementation is solely responsible for choosing the instance identifiers that differentiate instances within the instance domain. The PMDA is also responsible for ensuring the uniqueness of instance identifiers in any instance domain.

---

#### 2.3.4.1 N Dimensional Data

Where the performance data can be represented as scalar values (singular metrics) or one-dimensional arrays or lists (metrics with an instance domain), the PCP framework is more than adequate. In the case of metrics with an instance domain, each array or list element is associated with an instance from the instance domain.

To represent two or more dimensional arrays, the coordinates must be one of the following:

- Mapped onto one dimensional coordinates.
- Enumerated into the Performance Metrics Name Space (PMNS). For details, see Section 3.1, page 53.

For example, this 2 x 3 array of values called M can be represented as instances 1,..., 6 for a metric M:

```
M[1]   M[2]   M[3]
M[4]   M[5]   M[6]
```

Or they can be represented as instances 1, 2, 3 for metric M1 and instances 1, 2, 3 for metric M2:

```
M1[1]  M1[2]  M1[3]
M2[1]  M2[2]  M2[3]
```

The PMDA implementer must decide and consistently export this encoding from the N-dimensional instrumentation to the 1-dimensional data model of the PCP.

In certain special cases (for example, such as for a histogram), it may be appropriate to export an array of values as raw binary data (the type encoding in the descriptor is `PM_TYPEAggregate`). However, this requires the development of special PMAPI client tools, because the standard PCP tools have no knowledge of the structure and interpretation of the binary data.

#### 2.3.4.2 Data Structures

If the PMDA is required to support instance domains, then for each instance domain the unique internal instance identifier and external instance identifier should be defined using a `pmdaInstid` structure as shown in Example 2-7:

**Example 2-7** `pmdaInstid` Structure

```
typedef struct {
    int      i_inst;          /* internal instance identifier */
    char     *i_name;        /* external instance identifier */
} pmdaInstid;
```

The `i_inst` instance identifier must be a unique integer within a particular instance domain.

The complete instance domain description is specified in a `pmdaIndom` structure as shown in Example 2-8:

**Example 2-8** `pmdaIndom` Structure

```
typedef struct {
    pmInDom    it_indom;      /* indom, filled in */
    int        it_numinst;    /* number of instances */
    pmdaInstid *it_set;      /* instance identifiers */
} pmdaIndom;
```

The `it_indom` element contains a `pmInDom` that must be unique across every PMDA. The other fields of the `pmdaIndom` structure are the number of instances in the instance domain and a pointer to an array of instance descriptions.

Example 2-9 shows that the `pmInDom` can be safely cast to `_pmInDom_int`, which specifies the PMDA's domain and the instance number within the PMDA:

**Example 2-9** `_pmInDom_int` Structure

```
typedef struct {
    int          pad:2;
    unsigned int domain:8; /* the administrative PMD */
    unsigned int serial:22; /* unique within PMD */
} _pmInDom_int;
```

As with metrics, the PMDA domain number is not necessarily known until run time; so the domain field must be set up when the PMDA is initialized.

For information about how an instance domain may also be associated with more than one metric, see the `pmdaInit(3)` man page.

The simple PMDA, shown in Example 2-10, has five metrics and two instance domains of three instances.

**Example 2-10** Simple PMDA

```
/*
 * list of instances
 */
static pmdaInstid color[] = {
    { 0, ``red`` }, { 1, ``green`` }, { 2, ``blue`` }
};
static pmdaInstid    *timenow = NULL;
```

```

static unsigned int    timesize = 0;
/*
 * list of instance domains
 */
static pmdaIndom indomtab[] = {
#define COLOR_INDOM    0
    { COLOR_INDOM, 3, color },
#define NOW_INDOM      1
    { NOW_INDOM, 0, NULL },
};
/*
 * all metrics supported in this PMDA - one table entry for each
 */
static pmdaMetric metrictab[] = {
/* numfetch */
    { NULL,
      { PMDA_P MID(0,0), PM_TYPE_U32, PM_INDOM_NULL, PM_SEM_INSTANT,
        PMDA_PMUNITS ( 0,0,0,0,0,0) }, },
/* color */
    { NULL,
      { PMDA_P MID(0,1), PM_TYPE_32, COLOR_INDOM, PM_SEM_INSTANT,
        PMDA_PMUNITS ( 0,0,0,0,0,0) }, },
/* time.user */
    { NULL,
      { PMDA_P MID(1,2), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        PMDA_PMUNITS ( 0, 1, 0, 0, PM_TIME_SEC, 0 ) }, },
/* time.sys */
    { NULL,
      { PMDA_P MID(1,3), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        PMDA_PMUNITS ( 0, 1, 0, 0, PM_TIME_SEC, 0 ) }, },
/* now */
    { NULL,
      { PMDA_P MID(2,4), PM_TYPE_U32, NOW_INDOM, PM_SEM_INSTANT,
        PMDA_PMUNITS { 0,0,0,0,0,0 } }, },
};

```

The metric `simple.color` is associated, via `COLOR_INDOM`, with the first instance domain listed in `indomtab`. PMDA initialization assigns the correct domain portion of the instance domain identifier in `indomtab[0].it_indom` and `metrictab[1].m_desc.indom`. This instance domain has three instances: red, green, and blue.

The metric `simple.now` is associated, via `NOW_INDOM`, with the second instance domain listed in `indomtab`. PMDA initialization assigns the correct domain portion of the instance domain identifier in `indomtab[1].it_indom` and `metricstab[4].m_desc.indom`. This instance domain is dynamic and initially has no instances.

All other metrics are singular, as specified by `PM_INDOM_NULL`.

In some cases an instance domain may vary dynamically after PMDA initialization (for example, `simple.now`), and this requires some refinement of the default functions and data structures of the `libpcp_pmda` library. Briefly, this involves providing new functions that act as wrappers for `pmdaInstance` and `pmdaFetch` while understanding the dynamics of the instance domain, and then overriding the instance and fetch methods in the `pmdaInterface` structure during PMDA initialization.

For the simple PMDA, the wrapper functions are `simple_fetch` and `simple_instance`, and defaults are over-ridden by the following assignments in the `simple_init` function:

```
dp->version.two.fetch = simple_fetch;
dp->version.two.instance = simple_instance;
```

## 2.4 Other Issues

Other issues include extracting the information, latency and threads of control, Name Space , PMDA help text, and management of evolution within a PMDA.

### 2.4.1 Extracting the Information

A suggested approach to writing a PMDA is to write a standalone program to extract the values from the target domain and then incorporate this program into the PMDA framework. This approach avoids concurrent debugging of two distinct problems:

- Extraction of the data
- Communication with PMCD

These are some possible ways of exporting the data from the target domain:

- Accumulate the performance data in a public shared memory segment.



- Write the performance data to the end of a log file.
- Periodically rewrite a file with the most recent values for the performance data.
- Implement a protocol that allows a third party to connect to the target application, send a request, and receive new performance data.
- If the data is in the IRIX kernel, provide a system call (preferred) or global data (for a `/dev/kmem` reader) to export the performance data.

Most of these approaches require some further data processing by the PMDA.

## 2.4.2 Latency and Threads of Control

The PCP protocols expect PMDAs to return the current values for performance metrics when requested, and with short delay (low latency). For some target domains, access to the underlying instrumentation may be costly or involve unpredictable delays (for example, if the real performance data is stored on some remote host or network device). In these cases, it may be necessary to separate probing for new performance data from servicing PMCD requests.

An architecture that has been used successfully for several PMDAs is to create one or more `sproc` child processes to obtain information while the main process communicates with PMCD; see the `sproc(2)` man page. At the simplest deployment of this arrangement, the two processes may execute without synchronization. Pthreads have also been used as a more portable multithreading mechanism; see the `pthread(5)` man page.

By contrast, a complex deployment would be one in which the refreshing of the metric values must be atomic, and this may require double buffering of the data structures. It also requires coordination between parent and child processes.

---

**Tip:** Since PMAPI is not thread-safe, only one PMDA process or thread of control should call any PMAPI functions, and this would typically be the thread servicing requests from the PMCD.

---

One caveat about this style of caching PMDA—it is generally better if the PMDA converts counts to rates based upon consecutive periodic sampling from the underlying instrumentation. By exporting precomputed rate metrics with instantaneous semantics, the PMDA prevents the PCP monitor tools from computing their own rates upon consecutive PMCD fetches (which are likely to return identical values from a caching PMDA).

### 2.4.3 Name Space

The PMNS file defines the name space of the PMDA. It is a simple text file that is used during installation to expand the Name Space of the PMCD process. The format of this file is described by the `pmns(4)` man page.

Client processes will not be able to access the PMDA metrics if the PMNS file is not installed as part of the PMDA installation procedure on the collector host. The installed list of metric names and their corresponding PMIDs can be found in `/var/pcp/pmns/root`.

Example 2-11 shows the simple PMDA, which has five metrics:

- Three metrics immediately under the `simple` node
- Two metrics under another non-terminal node called `simple.time`

**Example 2-11** `pmns` File for the Simple PMDA

```
simple {
    numfetch    SIMPLE:0:0
    color       SIMPLE:0:1
    time
    now         SIMPLE:2:4
}
simple.time {
    user        SIMPLE:1:2
    sys         SIMPLE:1:3
}
```

Metrics that have different clusters do not have to be specified in different subtrees of the PMNS. Example 2-12 shows an alternative PMNS for the simple PMDA:

**Example 2-12** Alternate `pmns` File for the Simple PMDA

```
simple {
    numfetch    SIMPLE:0:0
    color       SIMPLE:0:1
    usertime    SIMPLE:1:2
    systime     SIMPLE:1:3
}
```

In this example, the `SIMPLE` macro is replaced by the domain number listed in `/var/pcp/pmns/stdpmid` for the corresponding PMDA during installation (for the simple PMDA, this would normally be the value 253).

## 2.4.4 PMDA Help Text

For each metric defined within a PMDA, the PMDA developer is strongly encouraged to provide both terse and extended help text to describe the metric, and perhaps provide hints about the expected value ranges.

The help text is used to describe each metric in the visualization tools and `pminfo` with the `-T` option. The help text, such as the help text for the simple PMDA in Example 2-13, is specified in a specially formatted file, normally called `help`. This file is converted to the expected run-time format using the `newhelp` command; see the `newhelp(1)` man page. Converted help text files are usually placed in the PMDA's directory below `/var/pcp/pmdas` as part of the PMDA installation procedure.

### Example 2-13 Help Text for the Simple PMDA

The two instance domains and five metrics have a short and a verbose description. Each entry begins with a line that starts with the character “@” and is followed by either the metric name (`simple.numfetch`) or a symbolic reference to the instance domain number (`SIMPLE.1`), followed by the short description. The verbose description is on the following lines, terminated by the next line starting with “@” or end of file:

```
@ SIMPLE.1 Instance domain ``colour`` for simple PMDA
Universally 3 instances, ``red`` (0), ``green`` (1) and ``blue`` (3).

@ SIMPLE.2 Dynamic instance domain ``time`` for simple PMDA
An instance domain is computed on-the-fly for exporting current time
information. Refer to the help text for simple.now for more details.

@ simple.numfetch Number of pmFetch operations.
The cumulative number of pmFetch operations directed to ``simple`` PMDA.

This counter may be modified with pmstore(1).

@ simple.color Metrics which increment with each fetch
This metric has 3 instances, designated ``red``, ``green`` and ``blue``.

The value of the metric is monotonic increasing in the range 0 to
```

255, then back to 0. The different instances have different starting values, namely 0 (red), 100 (green) and 200 (blue).

The metric values may be altered using `pmstore(1)`.

```
@ simple.time.user Time agent has spent executing user code
The time in seconds that the CPU has spent executing agent user code.
```

```
@ simple.time.sys Time agent has spent executing system code
The time in seconds that the CPU has spent executing agent system code.
```

```
@ simple.now Time of day with a configurable instance domain
The value reflects the current time of day through a dynamically
reconfigurable instance domain. On each metric value fetch request,
the agent checks to see whether the configuration file in
/var/pcp/pmdas/simple/simple.conf has been modified - if it has then
the file is re-parsed and the instance domain for this metric is again
constructed according to its contents.
```

This configuration file contains a single line of comma-separated time tokens from this set:

```
``sec`` (seconds after the minute),
``min`` (minutes after the hour),
``hour`` (hour since midnight).
```

An example configuration file could be: `sec,min,hour` and in this case the `simple.now` metric would export values for the three instances ```sec```, ```min``` and ```hour``` corresponding respectively to the components seconds, minutes and hours of the current time of day.

The instance domain reflects each token present in the file, and the values reflect the time at which the PMDA processes the fetch.

### 2.4.5 Management of Evolution within a PMDA

Evolution of a PMDA, or more particularly the underlying instrumentation to which it provides access, over time naturally results in the appearance of new metrics and the disappearance of old metrics. This creates potential problems for PMAPI clients and PCP tools that may be required to interact with both new and former versions of the PMDA.

The following guidelines are intended to help reduce the complexity of implementing a PMDA in the face of evolutionary change, while maintaining predictability and semantic coherence for tools using the PMAPI, and for end users of those tools.

- Try to support as full a range of metrics as possible in every version of the PMDA. In this context, *support* means responding sensibly to requests, even if the underlying instrumentation is not available.
- If a metric is not supported in a given version of the underlying instrumentation, the PMDA should respond to `pmLookupDesc` requests with a `pmDesc` structure whose `type` field has the special value `PM_TYPE_NOSUPPORT`. Values of fields other than `pmid` and `type` are immaterial, but Example 2-14 is typically benign:

**Example 2-14** Setting Values

```
pmDesc dummy = {
    PMDA_P MID(3,0),          /* pmid, fill this in */
    PM_TYPE_NOSUPPORT,      /* this is the important part */
    PM_INDOM_NULL,         /* singular, causes no problems */
    0,                      /* no semantics */
    { 0, 0, 0, 0, 0, 0 }    /* no units */
};
```

- If a metric lacks support in a particular version of the underlying instrumentation, the PMDA should respond to `pmFetch` requests with a `pmResult` in which no values are returned for the unsupported metric. This is marginally friendlier than the other semantically acceptable option of returning an illegal PMID error or `PM_ERR_P MID`.
- Help text should be updated with annotations to describe different versions of the underlying product, or product configuration options, for which a specific metric is available. This is so `pmLookupText` can always respond correctly.
- The `pmStore` operation should fail with return status of `-EACCES` if a user or application tries to amend the value of an unsupported metric.
- The value extraction, conversion, and printing functions (`pmExtractValue`, `pmConvScale`, `pmAtomStr`, `pmTypeStr`, and `pmPrintValue`) return the `PM_ERR_CONV` error or an appropriate diagnostic string, if an attempt is made to operate on a value for which `type` is `PM_TYPE_NOSUPPORT`.

If performance tools take note of the `type` field in the `pmDesc` structure, they should not manipulate values for unsupported metrics. Even if tools ignore `type` in the metric's description, following these development guidelines ensures that

no misleading value is ever returned; so there is no reason to call the extraction, conversion, and printing functions.

## 2.5 DSO Interface

This section describes an interface for the request handling callbacks in a PMDA. This interface is used by PMCD for communicating with DSO PMDAs, and can also be used by daemon PMDAs with `pmdaMain`.

### 2.5.1 Overview

Both daemon and DSO PMDAs must handle multiple request types from PMCD. A daemon PMDA communicates with PMCD using the PDU protocol, while a DSO PMDA defines callbacks for each request type. To avoid duplicating this PDU processing (in the case of a PMDA that can be installed either as a daemon or as a DSO), and to allow a consistent framework, `pmdaMain` can be used by a daemon PMDA as a wrapper to handle the communication protocol using the same callbacks as a DSO PMDA. This allows a PMDA to be built as both a daemon and a DSO, and then to be installed as either.

To further simplify matters, default callbacks are declared in `/usr/include/pcp/pmda.h`:

- `_pmInDom_int pmdaFetch`
- `pmdaProfile`
- `pmdaInstance`
- `pmdaDesc`
- `pmdaText`
- `pmdaStore`

Each callback takes a `pmdaExt` structure as its last argument. This structure contains all the information that is required by the default callbacks in most cases. The one exception is `pmdaFetch`, which needs an additional callback to instantiate the current value for each supported combination of a performance metric and an instance.

Therefore, for most PMDAs all the communication with PMCD is automatically handled by functions in `libpcp.so` and `libpcp_pmda.so`.

### 2.5.1.1 Trivial PMDA

The trivial PMDA uses all of the default callbacks as shown in Example 2-15. The additional callback for `pmdaFetch` is defined as `trivial_fetchCallback`:

**Example 2-15** Request Handling Callbacks in the Trivial PMDA

```
static int
trivial_fetchCallback(pmdaMetric *mdesc, unsigned int inst, pmAtomValue *atom)
{
    __pmID_int      *idp = (__pmID_int *)&(mdesc->m_desc.pmid);
    if (idp->cluster != 0 || idp->item != 0)
        return PM_ERR_PMIID;
    else if (inst != PM_IN_NULL)
        return PM_ERR_INST;
    atom->l = time(NULL);
    return 0;
}
```

This function checks that the PMID and instance are valid, and then places the metric value for the current time into the `pmAtomValue` structure. The callback is set up by a call to `pmdaSetFetchCallback` in `trivial_init`.

### 2.5.1.2 Simple PMDA

The simple PMDA callback for `pmdaFetch` is more complicated because it must support more metrics, some metrics are instantiated with each fetch, and one instance domain is dynamic. The default `pmdaFetch` callback, shown in Example 2-16, is replaced by `simple_fetch` in `simple_init`, which increments the number of fetches and updates the instance domain for `INDOM_NOW` before calling `pmdaFetch`:

**Example 2-16** Request Handling Callbacks in the Simple PMDA

```
static int
simple_fetch(int numpmid, pmID pmidlist[], pmResult **resp, pmdaExt *pmda)
{
    numfetch++;
    simple_timenow_check();
    simple_timenow_refresh();
    return pmdaFetch(numpmid, pmidlist, resp, pmda);
}
```

The callback for `pmdaFetch` is defined as `simple_fetchCallback`. The PMID is extracted from the `pmdaMetric` structure, and if valid, the appropriate field in the `pmAtomValue` structure is set.

The `simple.numfetch` metric has no instance domain and is easily handled first as shown in Example 2-17:

**Example 2-17** `simple.numfetch` Metric

```
static int
simple_fetchCallback(pmdaMetric *mdesc, unsigned int inst, pmAtomValue *atom)
{
    int i;
    static int oldfetch = 0;
    static struct tms tms;
    __pmID_int *idp = (__pmID_int *)&(mdesc->m_desc.pmid);
    if (inst != PM_IN_NULL &&
        !(idp->cluster == 0 && idp->item == 1) &&
        !(idp->cluster == 2 && idp->item == 4))
        return PM_ERR_INST;
    if (idp->cluster == 0) {
        if (idp->item == 0) {
            atom->l = numfetch; /* simple.numfetch */
        }
    }
}
```

In Example 2-18, the `inst` parameter is used to specify which instance is required for the `simple.color` metric:

**Example 2-18** `simple.color` Metric

```
else if (idp->item == 1) { /* simple.color */
    switch (inst) {
        case 0: /* red */
            red = (red + 1) % 256;
            atom->l = red;
            break;
        case 1: /* green */
            green = (green + 1) % 256;
            atom->l = green;
            break;
        case 2: /* blue */
            blue = (blue + 1) % 256;
            atom->l = blue;
    }
}
```



```

        break;
    default:
        return PM_ERR_INST;
    }
}
else
    return PM_ERR_P MID;

```

In Example 2-19, the `simple.time` metric is in a second cluster and has a simple optimization to reduce the overhead of calling `times` twice on the same fetch and return consistent values from a single call to `times` when both metrics `simple.time.user` and `simple.time.sys` are requested in a single `pmFetch`. The previous fetch count is used to determine if the `tms` structure should be updated:

**Example 2-19** `simple.time` Metric

```

else if (idp->cluster == 1) {                                /* simple.time */
    if (oldfetch < numfetch) {
        times(&tms);
        oldfetch = numfetch;
    }
    if (idp->item == 2)                                       /* simple.time.user */
        atom->d = (tms.tms_utime / (double)CLK_TCK);
    else if (idp->item == 3)                                  /* simple.time.sys */
        atom->d = (tms.tms_stime / (double)CLK_TCK);
    else
        return PM_ERR_P MID;
}

```

In Example 2-20, the `simple.now` metric is in a third cluster and uses `inst` again to select a specific instance from the `INDOM_NOW` instance domain:

**Example 2-20** `simple.now` Metric

```

else if (idp->cluster == 2) {
    if (idp->item == 4) {                                     /* simple.now */
        if (inst < timesize) {
            /* this loop will always match one of the named */
            /* time constants from the timeslices structure */
            for (i = 0; i < num_timeslices; i++) {
                if (strcmp(timenow[inst].i_name,
                    timeslices[i].tm_name) == 0) {
                    atom->l = timeslices[i].tm_field;
                }
            }
        }
    }
}

```

```

        break;
    }
}
else
    return PM_ERR_INST;
}
else
    return PM_ERR_P MID;
}

```

### 2.5.1.3 simple\_store in the Simple PMDA

The simple PMDA permits some of the metrics it supports to be modified by `pmStore` as shown in Example 2-21. For additional information, see the `pmstore(1)` man page.

#### Example 2-21 simple\_store in the Simple PMDA

The `pmdaStore` callback (which returns `-EACCESS` to indicate no metrics can be altered) is replaced by `simple_store` in `simple_init`. This replacement function must take the same arguments so that it can be assigned to the function pointer in the `pmdaInterface` structure.

The function traverses the `pmResult` and checks the cluster and unit of each PMID to ensure that it corresponds to a metric that can be changed. Checks are made on the values to ensure they are within range before being assigned to variables in the PMDA that hold the current values for exported metrics:

```

static int
simple_store(pmResult *result, pmdaExt *pmda)
{
    int          i, j, val, sts = 0;
    pmAtomValue av;
    pmValueSet  *vsp = NULL;
    __pmID_int  *pmidp = NULL;
    for (i = 0; i < result->numpmid; i++) {
        vsp = result->vset[i];
        pmidp = (__pmID_int *)&vsp->pmid;
        if (pmidp->cluster == 0) { /* storable metrics are cluster0 */
            switch (pmidp->item) {
                case 0: /* simple.numfetch */
                    val = vsp->vlist[0].value.lval;

```

```

    if (val < 0) {
        sts = PM_ERR_SIGN;
        val = 0;
    }
    numfetch = val;
    break;
case 1:
    /* simple.color */
    for (j = 0; j < vsp->numval && sts == 0; j++) {
        val = vsp->vlist[j].value.lval;
        if (val < 0) {
            sts = PM_ERR_SIGN;
            val = 0;
        } if (val > 255) {
            sts = PM_ERR_CONV;
            val = 255;
        }
    }

```

The `simple.color` metric has an instance domain that must be searched because any or all instances may be specified. Any instances that are not supported in this instance domain should cause an error value of `PM_ERR_INST` to be returned as shown in Example 2-22:

**Example 2-22** `simple.color` and `PM_ERR_INST` Errors

```

    switch (vsp->vlist[j].inst) {
        case 0:
            red = val;
            break;
            /* red */
        case 1:
            green = val;
            break;
            /* green */
        case 2:
            blue = val;
            break;
            /* blue */
        default:
            sts = PM_ERR_INST;
    }

```

Any other PMIDs in cluster 0 that are not supported by the simple PMDA should result in an error value of `PM_ERR_PMID` as shown in Example 2-23:

**Example 2-23** `PM_ERR_PMID` Errors

```

        default:
            sts = PM_ERR_PMID;
            break;
    }
}

```

Any metrics that cannot be altered should generate an error value of `-EACCES`, and metrics not supported by the PMDA should result in an error value of `PM_ERR_PMID` as shown in Example 2-24:

**Example 2-24** `-EACCES` and `PM_ERR_PMID` Errors

```

    else if ((pmidp->cluster == 1 &&
             (pmidp->item == 2 || pmidp->item == 3)) ||
            (pmidp->cluster == 2 && pmidp->item == 4)) {
        sts = -EACCES;
        break;
    }
    else {
        sts = PM_ERR_PMID;
        break;
    }
}
return sts;
}

```

The structure `pmdaExt` argument is not used by the `simple_store` function above.

#### 2.5.1.4 Return Codes for `pmdaFetch` Callbacks

In `PMDA_INTERFACE_1` and `PMDA_INTERFACE_2`, the return codes for the `pmdaFetch` callback function are defined:

Value	Meaning
< 0	Error code (for example, <code>PM_ERR_PMID</code> , <code>PM_ERR_INST</code> or <code>PM_ERR_AGAIN</code> )
0	Success

In `PMDA_INTERFACE_3`, the return codes for the `pmdaFetch` callback function are defined:

Value	Meaning
< 0	Error code (for example, <code>PM_ERR_PMID</code> , <code>PM_ERR_INST</code> )
0	Metric value not currently available
> 0	Success

## 2.5.2 PMDA Structures

PMDA structures used with the `pcp_pmda` library are defined in `/usr/include/pcp/pmda.h`. Example 2-25 and Example 2-26 describe the `pmdaInterface` and `pmdaExt` structures.

### Example 2-25 `pmdaInterface` Structure

The callbacks must be specified in a `pmdaInterface` structure:

```
typedef struct {
    int domain;        /* set/return performance metrics domain id here */
    struct {
        unsigned int pmda_interface : 8; /* PMDA DSO version */
        unsigned int pmapi_version : 8;  /* PMAPI version */
        unsigned int flags : 16;         /* usage TBD */
    } comm;           /* set/return communication and version info */
    int status;       /* return initialization status here */
    union {
/* Interface Version 1 (PCP 1.0 & PCP 1.1) */
        struct {
            int (*profile)(__pmProfile *);
            int (*fetch)(int, pmID *, pmResult **);
            int (*desc)(pmID, pmDesc *);
            int (*instance)(pmInDom, int, char *, __pmInResult **);
            int (*text)(int, int, char **);
            int (*control)(pmResult *, int, int, int);
            int (*store)(pmResult *);
        } one;
/*
 * Interface Version 2 (PCP 2.0) or later
 * PMDA_INTERFACE2, PMDA_INTERFACE3, ...
 */
    };
};
```

```

struct {
    pmdaExt *ext;
    int      (*profile)(__pmProfile *, pmdaExt *);
    int      (*fetch)(int, pmID *, pmResult **, pmdaExt *);
    int      (*desc)(pmID, pmDesc *, pmdaExt *);
    int      (*instance)(pmInDom, int, char *, __pmInResult **,
                        pmdaExt *);
    int      (*text)(int, int, char **, pmdaExt *);
    int      (*store)(pmResult *, pmdaExt *);
} two;
} version;

```

This structure is passed by PMCD to a DSO PMDA as an argument to the initialization function. This structure supports two versions—the second version adds support for the `pmdaExt` structure. Protocol version one is for backwards compatibility only, and should not be used in any new PMDA.

#### Example 2-26 `pmdaExt` Structure

Additional PMDA information must be specified in a `pmdaExt` structure:

```

typedef struct {
    unsigned int e_flags;           /* usage TBD */
    void         *e_ext;           /* usage TBD */
    char         *e_sockname;      /* socket name to pmcd */
    char         *e_name;         /* name of this pmda */
    char         *e_logfile;      /* path to log file */
    char         *e_helptext;     /* path to help text */
    int          e_status;        /* =0 is OK */
    int          e_infd;         /* input file descriptor from pmcd */
    int          e_outfd;        /* output file descriptor to pmcd */
    int          e_port;         /* port to pmcd */
    int          e_singular;     /* =0 for singular values */
    int          e_ordinal;      /* >=0 for non-singular values */
    int          e_direct;       /* =1 if pmid map to meta table */
    int          e_domain;       /* metrics domain */
    int          e_nmetrics;     /* number of metrics */
    int          e_nindoms;      /* number of instance domains */
    int          e_help;         /* help text comes via this handle */
    __pmProfile *e_prof;         /* last received profile */
    pmdaIoType   e_io;           /* connection type to pmcd */
    pmdaInDom    *e_indoms;      /* instance domain table */

```

```
pmdaIndom    *e_idp;          /* instance domain expansion */
pmdaMetric   *e_metrics;     /* metric description table */
pmdaResultCallBack e_resultCallBack; /* to clean up pmResult after fetch */
pmdaFetchCallBack e_fetchCallBack; /* to assign metric values in fetch */
pmdaCheckCallBack e_checkCallBack; /* callback on receipt of a PDU */
pmdaDoneCallBack e_doneCallBack; /* callback after PDU is processed */
} pmdaExt;
```

The `pmdaExt` structure contains filenames, pointers to tables, and some variables shared by several functions in the `pcp_pmda` library. All fields of the `pmdaInterface` and `pmdaExt` structures can be correctly set by PMDA initialization functions; see the `pmdaDaemon(3)`, `pmdaDSO(3)`, `pmdaGetOpt(3)`, `pmdaInit(3)`, and `pmdaConnect(3)` man pages for a full description of how various fields in these structures may be set or used by `pcp_pmda` library functions.

## 2.6 Initializing a PMDA

Several functions are provided to simplify the initialization of a PMDA. These functions, if used, must be called in a strict order so that the PMDA can operate correctly.

### 2.6.1 Overview

The initialization process for a PMDA involves opening help text files, assigning callback function pointers, adjusting the metric and instance identifiers to the correct domains, and much more. The initialization of a daemon PMDA also differs significantly from a DSO PMDA, since the `pmdaInterface` structure is initialized by `main` or the PMCD process, respectively.

### 2.6.2 Common Initialization

As described in Section 2.2.2, page 13, an initialization function is provided by a DSO PMDA and called by PMCD. Using the standard PMDA wrappers, the same function can also be used as part of the daemon PMDA initialization. This PMDA initialization function performs the following tasks:

- Assigning callback functions to the function pointer interface of `pmdaInterface`
- Assigning pointers to the metric and instance tables from `pmdaExt`

- Opening the help text files
- Assigning the domain number to the instance domains
- Correlating metrics with their instance domains

If the PMDA uses the common data structures defined for the `pcp_pmda` library, most of these requirements can be handled by the default `pmdaInit` function; see the `pmdaInit(3)` man page.

Because the initialization function is the only initialization opportunity for a DSO PMDA, the common initialization function should also perform any DSO-specific functions that are required. A default implementation of this functionality is provided by the `pmdaDSO` function; see the `pmdaDSO(3)` man page.

### 2.6.2.1 Trivial PMDA

Example 2-27 shows the trivial PMDA, which has no instances (that is, all metrics have singular values) and a single callback. This callback is for the `pmdaFetch` function called `trivial_fetchCallback`; see the `pmdaFetch(3)` man page:

**Example 2-27** Initialization in the Trivial PMDA

```
void trivial_init(pmdaInterface *dp)
{
    pmdaSetFetchCallback(dp, trivial_fetchCallback);
    pmdaInit(dp, NULL, 0,
             metrictab, sizeof(metrictab)/sizeof(metrictab[0]));
}
```

The trivial PMDA is always installed as a daemon PMDA.

### 2.6.2.2 Simple PMDA

In Example 2-28, the simple PMDA uses its own callbacks to handle `PDU_FETCH` and `PDU_RESULT` request PDUs (for `pmFetch` and `pmStore` operations respectively), as well as providing `pmdaFetch` with the callback `simple_fetchCallback`.

**Example 2-28** Initialization in the Simple PMDA

```
static int      isDSO = 1;                /* =0 I am a daemon */
void simple_init(pmdaInterface *dp)
{
    if (isDSO)
```



```
        pmdaDSO(dp, PMDA_INTERFACE_2, ``simple DSO``,
                ``/var/pcp/pmdas/simple/help``);
    if (dp->status != 0)
        return;
    dp->version.two.fetch = simple_fetch;
    dp->version.two.store = simple_store;
    dp->version.two.instance = simple_instance;
    pmdaSetFetchCallBack(dp, simple_fetchCallBack);
    pmdaInit(dp, indomtab, sizeof(indomtab)/sizeof(indomtab[0]),
            metricstab, sizeof(metricstab)/sizeof(metricstab[0]));
}
```

The simple PMDA may be installed either as a daemon PMDA or a DSO PMDA. The static variable *isDSO* indicates whether the PMDA is running as a DSO or as a daemon. A daemon PMDA should change the value of this variable to 0 in *main*.

### 2.6.3 Daemon Initialization

In addition to the initialization function that can be shared by a DSO and a daemon PMDA, a daemon PMDA must also meet the following requirements:

- Create the `pmdaInterface` structure that is passed to the initialization function
- Parse any command-line arguments
- Open a log file (a DSO PMDA uses PMCD's log file)
- Set up the IPC connection between the PMDA and the PMCD process
- Handle incoming PDUs

All these requirements can be handled by default initialization functions in the `pcp_pmda` library; see the `pmdaDaemon(3)`, `pmdaGetOpt(3)`, `pmdaOpenLog(3)`, `pmdaConnect(3)`, and `pmdaMain(3)` man pages.

The simple PMDA requires no additional command-line arguments other than those handled by `pmdaGetOpt` as shown in Example 2-29. For additional information, see the `pmdaGetOpt(3)` man page.

**Example 2-29** main in the Simple PMDA

```
int
main(int argc, char **argv)
{
    int          err = 0;
    int          c = 0;
    pmdaInterface dispatch;
    char         *p;
    /* trim cmd name of leading directory components */
    pmPrognome = argv[0];
    for (p = pmPrognome; *p; p++) {
        if (*p == '/')
            pmPrognome = p+1;
    }
    isDSO = 0;
    pmdaDaemon(&dispatch, PMDA_INTERFACE_2, pmPrognome, SIMPLE,
              ``simple.log``, ``/var/pcp/pmdas/simple/help``);
    if ((c = pmdaGetOpt(argc, argv, ``D:d:i:l:pu:??``, &dispatch, &err)) != EOF)
        err++;
    if (err)
        usage();
    pmdaOpenLog(&dispatch);
    simple_init(&dispatch);
    simple_timenow_check();
    pmdaConnect(&dispatch);
    pmdaMain(&dispatch);
    exit(0);
    /*NOTREACHED*/
}
```

## 2.7 Testing and Debugging a PMDA

Ensuring the correct operation of a PMDA can be difficult, because the responsibility of providing metrics to the requesting PMCD process and simultaneously retrieving values from the target domain requires nearly real-time communication with two

modules beyond the PMDA's control. Some tools are available to assist in this important task.

### 2.7.1 Overview

Thoroughly testing a PMDA with PMCD is difficult, although testing a daemon PMDA is marginally simpler than testing a DSO PMDA. If a DSO PMDA exits, PMCD also exits because they share a single address space and control thread. If the PMDA dumps core, `dbx` and related tools cannot reasonably explore the generated core image, which includes the PMCD image and any other active DSO PMDAs. For more information, see the `dbx(1)` man page.

The difficulty in using PMCD to test a daemon PMDA results from PMCD requiring timely replies from the PMDA in response to request PDUs. Although a timeout period can be set in `/etc/config/pccd.options`, attaching `dbx` to the PMDA process (or any other long delay) might cause an already running PMCD to close its connection with the PMDA. If timeouts are disabled, PMCD could wait forever to connect with the PMDA.

If you suspect a PMDA has been terminated due to a timeout failure, check the PMCD log file, usually `/var/adm/pcplog/pccd.log`.

A more robust way of testing a PMDA is to use the `dbpmda` tool, which is similar to PMCD except that `dbpmda` provides complete control over the PDUs that are sent to the PMDA, and there are no time limits—it is essentially an interactive debugger for exercising a PMDA. See the `dbpmda(3)` man page for details.

In addition, careful use of PCP debugging flags can produce useful information concerning a PMDA's behavior; see the `pmapi(3)` and `pmdbg(1)` man pages for a discussion of the PCP debugging and tracing framework.

### 2.7.2 Debugging Information

You can activate debugging flags in PMCD and most other PCP tools with the `-D` command-line option. Supported flags can be listed with the `pmdbg` command; see the `pmdbg(1)` man page. Setting the debug flag for PMCD in `/etc/config/pccd.options` might generate too much information to be useful, especially if there are other clients and PMDAs connected to the PMCD process.

The PMCD debugging flag can also be changed dynamically by storing a new value into the metric `pmcd.control.debug`:

```
# pmstore pmcd.control.debug 5
```

Most of the `pcp_pmda` library functions log additional information if the `DBG_TRACE_LIBPMDA` flag is set within the PMDA; see the `pmda(3)` man page. The command-line argument `-D` is trapped by `pmdaGetOpt` to set the global debugging control variable `pmDebug`. Adding tests within the PMDA for the `DBG_TRACE_APPL0`, `DBG_TRACE_APPL1`, and `DBG_TRACE_APPL2` trace flags permits different levels of information to be logged to the PMDA's log file.

All diagnostic, debugging, and tracing output from a PMDA should be written to the standard error stream. By convention, all debugging information is enclosed by preprocessor `#ifdef DEBUG` statements so that they can be compiled out of the program at a later stage, if required.

Adding this segment of code to the `simple_store` metric causes a log message to be sent to the current log file whenever `pmstore` attempts to change `simple.numfetch` and `pmDebug` has the `DBG_TRACE_APPL0` flag set as shown in Example 2-30:

**Example 2-30** `simple.numfetch` in the Simple PMDA

```
case 0: /* simple.numfetch */
    val = vsp->vlist[0].value.lval;
    if (val < 0) {
        sts = PM_ERR_SIGN;
        val = 0;
    }
#ifdef DEBUG
    if (pmDebug & DBG_TRACE_APPL0) {
        fprintf(stderr,
            "simple: %d stored into numfetch", val);
    }
#endif
    numfetch = val;
    break;
```

For a description of `pmstore`, see the `pmstore(1)` man page.

### 2.7.3 `dbpmda` Debug Utility

The `dbpmda` utility provides a simple interface to the PDU communication protocol. It allows daemon and DSO PMDAs to be tested with most request types, while the

PMDA process may be monitored with `dbx`, `par`, and other diagnostic tools. The `dbpmda(1)` man page contains a sample session with the simple PMDA.

## 2.8 Integration of PMDA

Several steps are required to install (or remove) a PMDA from a production PMCD environment without affecting the operation of other PMDAs or related visualization and logging tools.

The PMDA typically would have its own directory below `/var/pcp/pmdas` into which several files would be installed. In the description in Section 2.8.1, page 47, the PMDA of interest is assumed to be known by the name `newbie`, hence the PMDA directory would be `/var/pcp/pmdas/newbie`.

---

**Note:** Any installation or removal of a PMDA involves updating files and directories that are typically well protected. Hence the procedures described in this section must be executed as the superuser.

---

### 2.8.1 Installing a PMDA

A PMDA is fully installed when these tasks are completed:

- Help text has been installed in a place where the PMDA can find it, usually in the PMDA directory `/var/pcp/pmdas/newbie`.
- The name space has been updated in the `/var/pcp/pmns` directory.
- The PMDA binary has been installed, usually in the directory `/var/pcp/lib` for a DSO PMDA, or in the PMDA directory `/var/pcp/pmdas/newbie` for a daemon PMDA.
- The `/etc/pmcd.conf` file has been updated.
- The PMCD process has been restarted or notified (with a `SIGHUP` signal) that the new PMDA exists.

These tasks can be accomplished by a `Makefile` and an `Install` script as described below.

The `Makefile` should include an `install` target to compile and link the PMDA (as a DSO, or a daemon or both) in the PMDA directory, and in the case of a DSO

PMDA, install the shared library in `/var/pcp/lib`. The `clobber` target should remove any files created as a by-product of the `install` target.

You may wish to use `/var/pcp/pmdas/simple/Makefile` as a template for constructing a new PMDA Makefile; changing the assignment of `IAM` from `simple` to `newbie` would account for most of the required changes.

Since the object format of a DSO PMDA must match the object format of PMCD, which in turn must match the object format of the booted IRIX kernel, there might be multiple DSO targets in the Makefile. For example, see targets `mips_o32.pmda_$(IAM).so`, `mips_n32.pmda_$(IAM).so`, and `mips_64.pmda_$(IAM).so` for the simple PMDA.

The `Install` script should make use of the generic procedures defined in the script `/usr/pcp/lib/pmdaproc.sh`, and may be as straightforward as the one used for the trivial PMDA, shown in Example 2-31:

**Example 2-31** `Install` Script for the Trivial PMDA

```
# Get the common procedures and variable assignments
#
. /usr/pcp/lib/pmdaproc.sh
# The name of the PMDA
#
iam=trivial
# Do it
#
_setup
_install
exit 0
```

The variables, shown in Table 2-1, may be assigned values to modify the behavior of the `_setup` and `_install` procedures from `/usr/pcp/lib/pmdaproc.sh`.

**Table 2-1** Variables to Control Behavior of Generic `pmdaproc . sh` Procedures

Shell Variable	Use	Default
<code>\$iam</code>	Name of the PMDA; assignment to this variable is mandatory. Example: <code>iam=newbie</code>	
<code>\$dso_opt</code>	Can this PMDA be installed as a DSO?	<code>false</code>
<code>\$daemon_opt</code>	Can this PMDA be installed as a daemon?	<code>true</code>
<code>\$pipe_opt</code>	If installed as a daemon PMDA, is the default IPC via pipes?	<code>true</code>
<code>\$socket_opt</code>	If installed as a daemon PMDA, is the default IPC via an Internet socket?	<code>false</code>
<code>\$socket_inet_def</code>	If installed as a daemon PMDA, and the IPC method uses an Internet socket, the default port number.	
<code>\$ipc_prot</code>	IPC style for PDU exchanges involving a daemon PMDA; <code>binary</code> or <code>text</code> .	<code>binary</code>
<code>\$check_delay</code>	Delay in seconds between installing PMDA and checking if metrics are available.	<code>3</code>
<code>\$args</code>	Additional command-line arguments passed to a daemon PMDA.	
<code>\$pmda_interface</code>	Version of the <code>libpcp_pmda</code> library required, used to determine the version for generating help text files.	<code>1</code>
<code>\$pmns_source</code>	The name of the PMNS file (by default relative to the PMDA directory).	<code>pmns</code>
<code>\$pmns_name</code>	First-level name for this PMDA's metrics in the PMNS.	<code>\$iam</code>
<code>\$help_source</code>	The name of the help file (by default relative to the PMDA directory).	<code>help</code>
<code>\$pmda_name</code>	The name of the executable for a daemon PMDA.	<code>pmda\$iam</code>
<code>\$dso_name</code>	The name of the shared library for a DSO PMDA.	<code>pmda\$iam.so</code>

Shell Variable	Use	Default
<code>\$dso_entry</code>	The name of the initialization function for a DSO PMDA.	<code>\${iam}_init</code>
<code>\$domain</code>	The numerical PMDA domain number (from <code>domain.h</code> ).	
<code>\$SYMDOM</code>	The symbolic name of the PMDA domain number (from <code>domain.h</code> ).	

In addition, the variables `do_pmda` and `do_check` will be set to reflect the intention to install the PMDA (as opposed to install just the PMNS) and to check the availability of the metrics once the PMDA is installed. By default, each variable is `true`; however, the command-line options `-N` and `-Q` to `Install` may be used to set the variables to `false`, as follows: `do_pmda (-N)` and `do_check (-N or -Q)`.

The variables may also have their assignments changed by the user’s response to the common prompt as shown in Example 2-32:

**Example 2-32** Changing Variable Assignments

```
You will need to choose an appropriate configuration for installation
of the ... Performance Metrics Domain Agent (PMDA).
  collector   collect performance statistics on this system
  monitor    allow this system to monitor local and/or remote systems
  both       collector and monitor configuration for this system
```

Obviously, for anything but the most trivial PMDA, after calling the `_setup` procedure, the `Install` script should also prompt for any PMDA-specific parameters, which are typically accumulated in the `args` variable and used by the `_install` procedure.

The detailed operation of the `_install` procedure involves the following tasks:

- Using default assignments, and interaction where ambiguity exists, determine the PMDA type (DSO or daemon) and the IPC parameters, if any.
- Copy the `$pmns_source` file, replacing symbolic references to `SYMDOM` by the desired numeric domain number from `domain`.
- Merge the PMDA’s name space into the PCP name space at the non-leaf node identified by `$pmns_name`.



- If any `pmchart` views can be found (files with names ending in “.pmchart”), copy these to the standard directory (`/var/pcp/config/pmchart`) with the “.pmchart” suffix removed.
- Create new help files from `$help_source` after replacing symbolic references to `SYMDOM` by the desired numeric domain number from `domain`.
- Terminate the old daemon PMDA, if any.
- Use the `Makefile` to build the appropriate executables.
- Add the PMDA specification to PMCD's configuration file (`/etc/pccd.conf`).
- Notify PMCD. To minimize the impact on the services PMCD provides, sending a `SIGHUP` to PMCD forces it to reread the configuration file and start, restart, or remove any PMDAs that have changed since the file was last read.
- Check that the metrics from the new PMDA are available.

There are some PMDA changes that may trick PMCD into thinking nothing has changed, and not restarting the PMDA. Most notable are changes to the PMDA executable. In these cases, you may need to explicitly remove the PMDA as described in Section 2.8.3, page 51, or more drastically, restart PMCD as follows:

```
# /etc/init.d/pcp start
```

The files `/var/pcp/pmdas/*/Install` provide a wealth of examples that may be used to construct a new `PMDAInstall` script.

## 2.8.2 Upgrading a PMNS to Include Metrics from a New PMDA

When invoked with a `-N` command-line option, the `PMDA Install` script may be used to update the PMNS without installing the PMDA. This is typically used on a monitoring system to populate the local PMNS with the names of the performance metrics from a PMDA installed on a remote host running the older PCP 1.x protocols. The `-N` option also installs `pmchart` views useful on a monitoring system.

## 2.8.3 Removing a PMDA

The simplest way to stop a PMDA from running, apart from killing the process, is to remove the entry from `/etc/pccd.conf` and signal PMCD (with `SIGHUP`) to reread its configuration file. To completely remove a PMDA requires the reverse process of the installation, including an update of the Performance Metrics Name Space (PMNS).

This typically involves a `Remove` script in the PMDA directory that uses the same common procedures as the `Install` script described Section 2.8.1, page 47.

The `/var/pcp/pmdas/*/Remove` files provide a wealth of examples that may be used to construct a new PMDA `Remove` script.

## 2.8.4 Configuring PCP Tools

Most PCP tools have their own configuration file format for specifying which metrics to view or to log. By using canned configuration files that monitor key metrics of the new PMDA, users can quickly see the performance of the target system, as characterized by key metrics in the new PMDA.

Any configuration files that are created should be kept with the PMDA and installed into the appropriate directories when the PMDA is installed.

The `pmchart` command comes with several views for the default PMDAs located at `/var/pcp/config/pmchart`; see the `pmchart(1)` man page. These views can be used as a basis for defining views relevant to the new PMDA.

Likewise, there are several shell scripts that employ `pmview` for 3-dimensional visualizations, including `dkvis` and `mpvis`; see the `pmview(1)`, `dkvis(1)` and `mpvis(1)` man pages. Only small sections of these scripts require modification to visualize a different set of metrics. Similar scripted front ends could be created to customize `pmgadgets` icon control panels for a new PMDA; refer to the `pmsys()` man page.

As with all PCP customization, some of the most valuable tools can be created by defining views, scenes, and control-panel layouts that combine related performance metrics from multiple PMDAs or multiple hosts.

Parameterized alarm configurations can be created using the `pmieconf` facilities; see the `pmieconf(1)` and `pmie(1)` man pages. In addition, `pmie` rules involving metrics from the new PMDA may be created directly.

Daily logs can be specified in `pmlogger` configuration files, or with the `pmlogger_daily` mechanism; see the `pmlogger(1)` and `pmlogger_daily(1)` man pages. The services of `pmsnap` may be used to incorporate the new performance metrics into charts that may be periodically regenerated and published via a World Wide Web server.

## PMAPI—The Performance Metrics API

This chapter describes the Performance Metrics Application Programming Interface (PMAPI) provided with Performance Co-Pilot (PCP).

The PMAPI is a set of functions and data structure definitions that allow client applications to access performance data from one or more Performance Metrics Collection Daemons (PMCDs) or from PCP archive logs. The PCP utilities are all written using the PMAPI.

The most common use of PCP includes running performance monitoring utilities on a workstation (the monitoring system) while performance data is retrieved from one or more remote collector systems by a number of PCP processes. These processes execute on both the monitoring system and the collector systems. The collector systems are typically servers, and are the targets for the performance investigations.

In the development of the PMAPI the most important question has been, “How easily and quickly will this API enable the user to build new performance tools, or exploit existing tools for newly available performance metrics?” The PMAPI and the standard tools that use the PMAPI have enjoyed a symbiotic evolution throughout the development of PCP.

It will be convenient to differentiate between code that uses the PMAPI and code that implements the services of the PMAPI. The former will be termed “above the PMAPI” and the latter “below the PMAPI.”

### 3.1 Naming and Identifying Performance Metrics

Across all of the supported performance metric domains, there are a large number of performance metrics. Each metric has its own description, format, and semantics. PCP presents a uniform interface to these metrics above the PMAPI, independent of the source of the underlying metric data. For example, the performance metric `hinv.physmem` has a single 32-bit unsigned integer value, representing the number of megabytes of physical memory in the system, while the performance metric `disk.dev.total` has one 32-bit unsigned integer value per disk spindle, representing the cumulative count of I/O operations involving each associated disk spindle. These concepts are described in greater detail in Section 2.3, page 16.

For brevity and efficiency, internally PCP avoids using ASCII names for performance metrics, and instead uses an identification scheme that unambiguously associates a

single integer with each known performance metric. This integer is known as a Performance Metric Identifier, or PMID. For functions using the PMAPI, a PMID is defined and manipulated with the typedef `pmID`.

Below the PMAPI, the integer value of the PMID has an internal structure that reflects the details of the PMCD and PMDA architecture, as described in Section 2.3.3, page 18.

Above the PMAPI, a Performance Metrics Name Space (PMNS) is used to provide a hierarchic classification of external metric names, and a one-to-one mapping of external names to internal PMIDs. A more detailed description of the PMNS can be found in the *Performance Co-Pilot User's and Administrator's Guide*.

Applications that use the PMAPI may have independent versions of a PMNS, constructed from an initialization file when the application starts. Not all PMIDs need be represented in the PMNS of every application. For example, an application that monitors disk traffic could use a Name Space that references only the PMIDs for I/O statistics. Other applications require a stable PMNS that can be assumed to be the same on all systems. The distributed implementation includes a default PMNS for just this purpose.

The vast majority of PCP users and applications using the PMAPI will choose to use the default PMNS.

As of PCP release 2.0 the default PMNS comes from the performance metrics source, either a PMCD process or a PCP archive. This PMNS always reflects the available metrics from the performance metrics source; so most applications never use the local version of a PMNS.

## 3.2 Performance Metric Instances

When performance metric values are returned across the PMAPI to a requesting application, there may be more than one value for a particular metric; for example, independent counts for each CPU, or each process, or each disk, or each system call type, and so on. This multiplicity of values is not enumerated in the Name Space, but rather when performance metrics are delivered across the PMAPI.

The notion of `metric instances` is really a number of related concepts, as follows:

- A particular performance metric may have a set of associated values or instances.
- The instances are differentiated by an instance identifier.

- An instance identifier has an internal encoding (an integer value) and an external encoding (a corresponding external name or label).
- The set of all possible instance identifiers associated with a performance metric on a particular host constitutes an *instance domain*.
- Several performance metrics may share the same instance domain.

Consider Example 3-1:

**Example 3-1** Metrics Sharing the Same Instance Domain

```
$ pminfo -f filesystems.free
```

```
filesystems.free
  inst [1 or ``/dev/root``] value 1803
  inst [2 or ``/dev/usr``] value 22140
  inst [3 or ``/dev/dsk/dks0d2s0``] value 157938
```

The metric `filesystems.free` has three values, currently 1803, 22140, and 157938. These values are respectively associated with the instances identified by the internal identifiers 1, 2 and 3, and the external identifiers `/dev/root`, `/dev/usr`, and `/dev/dsk/dks0d2s0`. These instances form an instance domain that is shared by the performance metrics `filesystems.capacity`, `filesystems.used`, `filesystems.free`, `filesystems.mountdir`, and so on.

Each performance metric is associated with an instance domain, while each instance domain may be associated with many performance metrics. Each instance domain is identified by a unique value, as defined by the following typedef declaration:

```
typedef unsigned long pmInDom;
```

The special instance domain `PM_INDOM_NULL` is reserved to indicate that the metric has a single value (a singular instance domain). For example, the performance metric `mem.freemem` always has exactly one value. Note that this is semantically different to a performance metric like `kernel.percpu.syscall` that has a non-singular instance domain, but may have only one value available; for example, on a system with a single processor.

In the results returned above the PMAPI, each individual instance, within an instance domain, is identified by an internal integer instance identifier. The special instance identifier `PM_IN_NULL` is reserved for the single value in a singular instance domain. Performance metric values are delivered across the PMAPI as a set of instance identifier and value pairs.

The instance domain of a metric may change with time. For example, a machine may be shut down, have several disks added, and be rebooted. All performance metrics associated with the instance domain of disk devices would contain additional values after the reboot. The difficult issue of transient performance metrics means that repeated requests for the same PMID may return different numbers of values, or some changes in the particular instance identifiers returned. This means applications need to be aware that metric instantiation is guaranteed to be valid only at the time of collection.

---

**Note:** Some instance domains are more dynamic than others. For example, consider the instance domains behind the performance metrics `proc.memory.physical.dat` (one instance per process), `swap.free` (one instance per swap partition) and `kernel.percpu.cpu.intr` (one instance per CPU).

---

### 3.3 Current PMAPI Context

When performance metrics are retrieved across the PMAPI, they are delivered in the context of a particular source of metrics, a point in time, and a profile of desired instances. This means that the application making the request has already negotiated across the PMAPI to establish the context in which the request should be executed.

A metric's source may be the current performance data from a particular host (a live or real-time source), or an archive log of performance data collected by `pmlogger` at some remote host or earlier time (a retrospective or archive source). The metric's source is specified when the PMAPI context is created by calling the `pmNewContext` function.

The collection time for a performance metric is always the current time of day for a real-time source, or current position for an archive source. For archives, the collection time may be set to an arbitrary time within the bounds of the archive log by calling the `pmSetMode` function.

The last component of a PMAPI context is an instance profile that may be used to control which particular instances from an instance domain should be retrieved. When a new PMAPI context is created, the initial state expresses an interest in all possible instances, to be collected at the current time. The instance profile can be manipulated using the `pmAddProfile` and `pmDelProfile` functions.

### 3.4 Performance Metric Descriptions

For each defined performance metric, there is associated metadata encoded in a performance metric description (`pmDesc` structure) that describes the format and semantics of the performance metric. The `pmDesc` structure, in Example 3-2, provides all of the information required to interpret and manipulate a performance metric through the PMAPI. It has the following declaration:

**Example 3-2** `pmDesc` Structure

```
/* Performance Metric Descriptor */
typedef struct {
    pmID    pmid; /* unique identifier */
    int     type; /* base data type (see below) */
    pmInDom indom; /* instance domain */
    int     sem; /* semantics of value (see below) */
    pmUnits units; /* dimension and units (see below) */
} pmDesc;
```

The `type` field in the `pmDesc` structure describes various encodings of a metric's value. Its value will be one of the following constants:

```
/* pmDesc.type - data type of metric values */
#define PM_TYPE_NOSUPPORT -1 /* not in this version */
#define PM_TYPE_32      0 /* 32-bit signed integer */
#define PM_TYPE_U32     1 /* 32-bit unsigned integer */
#define PM_TYPE_64      2 /* 64-bit signed integer */
#define PM_TYPE_U64     3 /* 64-bit unsigned integer */
#define PM_TYPE_FLOAT   4 /* 32-bit floating point */
#define PM_TYPE_DOUBLE  5 /* 64-bit floating point */
#define PM_TYPE_STRING  6 /* array of char */
#define PM_TYPE_AGGREGATE 7 /* arbitrary binary data */
#define PM_TYPE_AGGREGATE_STATIC 8 /* static pointer to aggregate */
#define PM_TYPE_UNKNOWN 255 /* used om pmValueBlock, not pmDesc */
```

By convention `PM_TYPE_STRING` is interpreted as a classic C-style null byte terminated string.

If the value of a performance metric is of type `PM_TYPE_AGGREGATE` (or indeed `PM_TYPE_STRING`), the interpretation of that value is unknown to most PCP components. In these cases, the application using the value and the Performance Metrics Domain Agent (PMDA) providing the value must have some common understanding about how the value is structured and interpreted.

PM\_TYPE\_NOSUPPORT indicates that the PCP collection framework knows about the metric, but the corresponding service or application is either not configured or is at a revision level that does not provide support for this performance metric.

The semantics of the performance metric is described by the `sem` field of a `pmDesc` structure and uses the following constants:

```
/* pmDesc.sem - semantics of metric values */
#define PM_SEM_COUNTER 1 /* cumulative count, monotonic increasing */
#define PM_SEM_INSTANT 3 /* instant. value continuous domain */
#define PM_SEM_DISCRETE 4 /* instant. value discrete domain */
```

Each value for a performance metric is assumed to be drawn from a set of values that can be described in terms of their dimensionality and scale by a compact encoding, as follows:

- The dimensionality is defined by a power, or index, in each of three orthogonal dimensions: Space, Time, and Count (dimensionless). For example, I/O throughput is  $\text{Space}^1 \cdot \text{Time}^{-1}$ , while the running total of system calls is  $\text{Count}^1$ , memory allocation is  $\text{Space}^1$ , and average service time per event is  $\text{Time}^1 \cdot \text{Count}^{-1}$ .
- In each dimension, a number of common scale values are defined that may be used to better encode ranges that might otherwise exhaust the precision of a 32-bit value. For example, a metric with dimension  $\text{Space}^1 \cdot \text{Time}^{-1}$  may have values encoded using the scale megabytes per second.

This information is encoded in the `pmUnits` data structure, shown in Example 3-3. It is embedded in the `pmDesc` structure :

**Example 3-3** `pmUnits` and `pmDesc` Structures

```
/*
 * Encoding for the units (dimensions and
 * scale) for Performance Metric Values
 *
 * For example, a pmUnits struct of
 * { 1, -1, 0, PM_SPACE_MBYTE, PM_TIME_SEC, 0 }
 * represents Mbytes/sec, while
 * { 0, 1, -1, 0, PM_TIME_HOUR, 6 }
 * represents hours/million-events
 */
typedef struct {
    int dimSpace:4; /* space dimension */
    int dimTime:4; /* time dimension */
```



```

        int dimCount:4; /* event dimension */
        int scaleSpace:4; /* one of PM_SPACE_* below */
        int scaleTime:4; /* one of PM_TIME_* below */
        int scaleCount:4; /* one of PM_COUNT_* below */
    } pmUnits; /* dimensional units and scale of value */
/* pmUnits.scaleSpace */
#define PM_SPACE_BYTE 0 /* bytes */
#define PM_SPACE_KBYTE 1 /* Kilobytes (1024) */
#define PM_SPACE_MBYTE 2 /* Megabytes (1024^2) */
#define PM_SPACE_GBYTE 3 /* Gigabytes (1024^3) */
#define PM_SPACE_TBYTE 4 /* Terabytes (1024^4) */
/* pmUnits.scaleTime */
#define PM_TIME_NSEC 0 /* nanoseconds */
#define PM_TIME_USEC 1 /* microseconds */
#define PM_TIME_MSEC 2 /* milliseconds */
#define PM_TIME_SEC 3 /* seconds */
#define PM_TIME_MIN 4 /* minutes */
#define PM_TIME_HOUR 5 /* hours */
/*
 * pmUnits.scaleCount (e.g. count events, syscalls,
 * interrupts, etc.) -- these are simply powers of 10,
 * and not enumerated here.
 * e.g. 6 for 10^6, or -3 for 10^-3
 */
#define PM_COUNT_ONE 0 /* 1 */

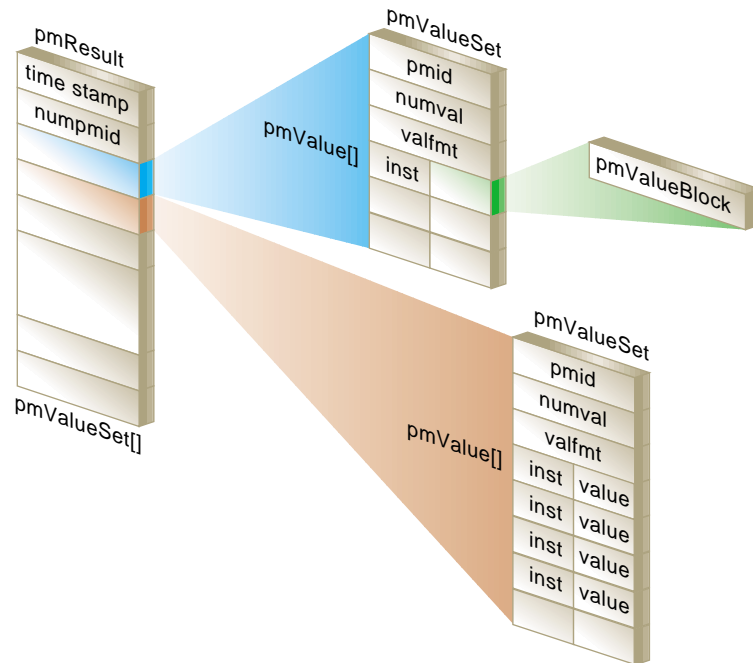
```

### 3.5 Performance Metrics Values

An application may fetch (or store) values for a set of performance metrics, each with a set of associated instances, using a single `pmFetch` (or `pmStore`) function call. To accommodate this, values are delivered across the PMAPI in the form of a tree data structure, rooted at a `pmResult` structure. This encoding is illustrated in Figure 3-1, and uses the component data structures in Example 3-4:

**Example 3-4** pmValueBlock and pmValue Structures

```
typedef struct {
    int inst; /* instance identifier */
    union {
        pmValueBlock *pval; /* pointer to value-block */
        int lval; /* integer value insitu */
    } value;
} pmValue;
```



**Figure 3-1** A Structured Result for Performance Metrics from pmFetch

The internal instance identifier is stored in the `inst` element. If a value for a particular metric-instance pair is a 32-bit integer (signed or unsigned), then it will be stored in the `lval` element. If not, the value will be in a `pmValueBlock` structure, as shown in Example 3-5, and will be located via `pval`:

**Example 3-5** pmValueBlock Structure

```
typedef struct {
    unsigned int    vtype : 8;    /* value type */
    unsigned int    vlen : 24;    /* bytes for vtype/vlen + vbuf */
    char            vbuf[1];      /* the value */
} pmValueBlock;
```

The length of the pmValueBlock (including the vtype and vlen fields) is stored in vlen. Despite the prototype declaration of vbuf, this array really accommodates vlen minus sizeof(vlen) bytes. The vtype field encodes the type of the value in the vbuf[] array, and is one of the PM\_TYPE\_\* macros defined in /usr/include/pmapi.h.

A pmValueSet structure, as shown in Example 3-6, contains all of the values to be returned from pmFetch for a single performance metric identified by the pmid field.

**Example 3-6** pmValueSet Structure

```
typedef struct {
    pmID    pmid;        /* metric identifier */
    int     numval;      /* number of values */
    int     valfmt;      /* value style, insitu or ptr */
    pmValue vlist[1];    /* set of instances/values */
} pmValueSet;
```

If positive, the numval field identifies the number of value-instance pairs in the vlist array (despite the prototype declaration of size 1). If numval is zero, there are no values available for the associated performance metric and vlist[0] is undefined. A negative value for numval indicates an error condition (see the pmErrStr(3) man page) and vlist[0] is undefined. The valfmt field has the value PM\_VAL\_INSITU to indicate that the values for the performance metrics should be located directly via the lval member of the value union embedded in the elements of vlist; otherwise, metric values are located indirectly via the pval member of the elements of vlist.

The pmResult structure, as shown in Example 3-7, contains a time stamp and an array of numpmid pointers to pmValueSet.

**Example 3-7** pmResult Structure

```
/* Result returned by pmFetch() */
typedef struct {
    struct timeval timestamp; /* stamped by collector */
    int          numpmid;    /* number of PMIDs */
    pmValueSet   *vset[1];  /* set of value sets */
} pmResult
```

There is one pmValueSet pointer per PMID, with a one-to-one correspondence to the set of requested PMIDs passed to pmFetch.

Along with the metric values, the PMAPI returns a time stamp with each pmResult that serves to identify when the performance metric values were collected. The time is in the format returned by gettimeofday and is typically very close to the time when the metrics are exported across the PMAPI.

---

**Note:** There is a question of exactly when individual metrics may have been collected, especially given their origin in potentially different performance metric domains, and variability in metric updating frequency by individual PMDAs. PCP uses a pragmatic approach, in which the PMAPI implementation returns all metrics with values accurate as of the time stamp, to the maximum degree possible, and PMCD demands that all PMDAs deliver values within a small realtime window. The resulting inaccuracy is small, and the additional burden of accurate individual timestamping for each returned metric value is neither warranted nor practical (from an implementation viewpoint).

---

The PMAPI provides functions to extract, rescale, and print values from the above structures; refer to Section 3.8, page 91.

### 3.6 PMAPI Programming Style and Interaction

The following sections describe the PMAPI programming style:

- Variable length argument and results lists
- PMAPI error handling

### 3.6.1 Variable Length Argument and Results Lists

All arguments and results involving a “list of something” are encoded as an array with an associated argument or function value to identify the number of elements in the array. This encoding scheme avoids both the `varargs` approach and sentinel-terminated lists. Where the size of a result is known at the time of a call, it is the caller's responsibility to allocate (and possibly free) the storage, and the called function assumes that the resulting argument is of an appropriate size.

Where a result is of variable size and that size cannot be known in advance (for example, `pmGetChildren`, `pmGetInDom`, `pmNameInDom`, `pmNameID`, `pmLookupText`, and `pmFetch`), the underlying implementation uses dynamic allocation through `malloc` in the called function, with the caller responsible for subsequently calling `free` to release the storage when no longer required. In the case of the result from `pmFetch`, there is a function (`pmFreeResult`) to release the storage, due to the complexity of the data structure and the need to make multiple calls to `free` in the correct sequence. As a general rule, if the called function returns an error status, then no allocation is done, the pointer to the variable sized result is undefined, and `free` or `pmFreeResult` should not be called.

### 3.6.2 PMAPI Error Handling

Where error conditions may arise, the functions that compose the PMAPI conform to a single, simple error notification scheme, as follows:

- The function returns an `int`. Values greater than or equal to zero indicate no error, and perhaps some positive status: for example, the number of items processed.
- Values less than zero indicate an error, as determined by a global table of error conditions and messages.

A PMAPI library function along the lines of `strerror` is provided to translate error conditions into error messages; see the `pmErrStr(3)` man page. The error condition is returned as the function value from a previous PMAPI call; there is no global error indicator (unlike `errno`). This is an attempt to anticipate and accommodate a programming environment that does not hinder the implementation of multi-threaded performance tools. The available error codes may be displayed with the following command:

```
pmerr -l
```

## 3.7 PMAPI Procedural Interface

The following sections describe all of the PMAPI functions that provide access to the PCP infrastructure on behalf of a client application:

- PMAPI Name Space services
- PMAPI metric description services
- PMAPI instance domain services
- PMAPI context services
- PMAPI timezone services
- PMAPI metrics services
- PMAPI record-mode services
- PMAPI archive-specific services
- PMAPI time control services
- PMAPI ancillary support services

### 3.7.1 PMAPI Name Space Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) Name Space services.

#### 3.7.1.1 `pmGetChildren` Function

```
int pmGetChildren(const char *name, char ***offspring)
```

Given a full pathname to a node in the current PMNS, as identified by *name*, return through *offspring* a list of the relative names of all the immediate descendents of *name* in the current PMNS. As a special case, if *name* is an empty string, (that is, "" but not NULL or (char \*)0), the immediate descendents of the root node in the PMNS are returned.

Normally, `pmGetChildren` returns the number of descendent names discovered, or a value less than zero for an error. The value zero indicates that the *name* is valid, and associated with a leaf node in the PMNS.

The resulting list of pointers (*offspring*) and the values (relative metric names) that the pointers reference are allocated by `pmGetChildren` with a single call to `malloc`, and it is the responsibility of the caller to issue a `free(offspring)` system call to release the space when it is no longer required. When the result of `pmGetChildren` is less than one, *offspring* is undefined (no space is allocated, and so calling `free` is counterproductive).

### 3.7.1.2 `pmGetChildrenStatus` Function

```
int  
pmGetChildrenStatus(const char *name, char ***offspring, int **status)
```

The `pmGetChildrenStatus` function is an extension of `pmGetChildren` that optionally returns status information about each of the descendent names.

Given a fully qualified pathname to a node in the current PMNS, as identified by *name*, `pmGetChildrenStatus` returns by means of *offspring* a list of the relative names of all of the immediate descendent nodes of *name* in the current PMNS. If *name* is the empty string (""), it returns the immediate descendents of the root node in the PMNS.

If *status* is not NULL, then `pmGetChildrenStatus` also returns the status of each child by means of *status*. This refers to either a leaf node (with value `PMNS_LEAF_STATUS`) or a non-leaf node (with value `PMNS_NONLEAF_STATUS`).

Normally, `pmGetChildrenStatus` returns the number of descendent names discovered, or else a value less than zero to indicate an error. The value zero indicates that *name* is a valid metric name, being associated with a leaf node in the PMNS.

The resulting list of pointers (*offspring*) and the values (relative metric names) that the pointers reference are allocated by `pmGetChildrenStatus` with a single call to `malloc`, and it is the responsibility of the caller to `free(offspring)` to release the space when it is no longer required. The same holds true for the *status* array.

### 3.7.1.3 `pmGetPMNSLocation` Function

```
int pmGetPMNSLocation(void)
```

If an application needs to know where the origin of a PMNS, `pmGetPMNSLocation` returns whether it is an archive (`PMNS_ARCHIVE`), a local PMNS file (`PMNS_LOCAL`), or a remote PMCD (`PMNS_REMOTE`). This information may be useful in determining an appropriate error message depending on PMNS location.

#### 3.7.1.4 pmLoadNameSpace Function

```
int pmLoadNameSpace(const char *filename)
```

Before requesting any services involving a local Performance Metrics Name Space (PMNS), the application must load the PMNS using `pmLoadNameSpace`.

The *filename* argument designates the PMNS of interest. For applications that do not require a tailored Name Space, the special value `PM_NS_DEFAULT` may be used for *filename*, to force a default local PMNS to be established. Externally, a PMNS may be stored in either an ASCII or binary format. The `pmnscomp` utility is used to create the binary format from the ASCII format.

---

**Note:** The distributed PMNS services in PCP 2.x avoid the need for a local PMNS in most cases; so applications typically would **not** use `pmLoadNameSpace`. If applications do not call `pmLoadNameSpace`, the default PMNS is the one at the source of the performance metrics.

---

#### 3.7.1.5 pmLoadASCIINameSpace Function

```
int pmLoadASCIINameSpace(const char *filename, int dupok)
```

If the application wants to force using a local Performance Metrics Name Space (PMNS) instead of a distributed PMNS, it must load the PMNS using `pmLoadASCIINameSpace` or `pmLoadNameSpace`. If the application wants to use a distributed PMNS, then it should not make a call to load the PMNS explicitly.

`pmLoadASCIINameSpace` is a variant of `pmLoadNameSpace`, which only processes an ASCII format PMNS. The *dupok* argument may be used to control the handling of multiple names in the PMNS that may be associated with a single Performance Metric Identifier (PMID). A value of 0 disallows duplicates; any other value allows duplicates.

The *filename* argument designates the PMNS of interest. For applications not requiring a tailored PMNS, the special value `PM_NS_DEFAULT` may be used for *filename*, to force the default local PMNS to be loaded. Since this PMNS exists in a binary format, `pmLoadNameSpace` is the more efficient function to use.

The default local PMNS is found in the `/var/pcp/pmns/root` file unless the `PMNS_DEFAULT` environment variable is set. Then the value is assumed to be the pathname to the file containing the default local PMNS.

`pmLoadASCIINameSpace` returns zero on success.



Syntax and other errors in the parsing of the PMNS are reported on `stderr` with a message of the form:

```
Error Parsing ASCII PMNS: ....
```

`PM_ERR_DUPPMNS` is an error to try and load more than one PMNS, or to call either `pmLoadASCIINamespace` or `pmLoadNamespace` more than once. `PM_ERR_PMNS` indicates a syntax error in an ASCII format PMNS.

### 3.7.1.6 `pmLookupName` Function

```
int pmLookupName(int numpmid, char *namelist[], pmID pmidlist[])
```

Given a list in *namelist* containing *numpmid* full pathnames for performance metrics from the current PMNS, `pmLookupName` returns the list of associated PMIDs through the *pmidlist* parameter. Invalid metrics names are translated to the error PMID value of `PM_ID_NULL`.

The result from `pmLookupName` is the number of names translated in the absence of errors, or an error indication. Note that argument definition and the error protocol guarantee a one-to-one relationship between the elements of *namelist* and *pmidlist*; both lists contain exactly *numpmid* elements.

### 3.7.1.7 `pmNameAll` Function

```
int pmNameAll(pmID pmid, char ***nameset)
```

Given a performance metric ID in *pmid*, `pmNameAll` determines all the corresponding metric names, if any, in the PMNS, and returns these through *nameset*.

The resulting list of pointers *nameset* and the values (relative names) that the pointers reference are allocated by `pmNameAll` with a single call to `malloc`. It is the caller's responsibility to call `free` and release the space when it is no longer required.

In the absence of errors, `pmNameAll` returns the number of names in *nameset*.

For many PMNS instances, there is a 1:1 mapping between a name and a PMID, and under these circumstances, `pmNameID` provides a simpler interface in the absence of duplicate names for a particular PMID.

### 3.7.1.8 pmNameID Function

```
int pmNameID(pmID pmid, char **name)
```

Given a performance metric ID in *pmid*, `pmNameID` determines the corresponding metric name, if any, in the current PMNS, and returns this through *name*.

In the absence of errors, `pmNameID` returns zero. The *name* argument is a null byte-terminated string, allocated by `pmNameID` using `malloc`. It is the caller's responsibility to call `free` and release the space when it is no longer required.

### 3.7.1.9 pmTraversePMNS Function

```
int pmTraversePMNS(const char *name, void (*dometric)(char *))
```

The function `pmTraversePMNS` may be used to perform a depth-first traversal of the PMNS. The traversal starts at the node identified by *name*—if *name* is an empty string, the traversal starts at the root of the PMNS. Usually, *name* would be the pathname of a non-leaf node in the PMNS.

For each leaf node (actual performance metrics) found in the traversal, the user-supplied function *dometric* is called with the full pathname of that metric in the PMNS as the single argument; this argument is a null byte-terminated string, and is constructed from a buffer that is managed internally to `pmTraversePMNS`. Consequently, the value is valid only during the call to *dometric*—if the pathname needs to be retained, it should be copied using `strdup` before returning from *dometric*; see the `strdup(3C)` man page.

### 3.7.1.10 pmTrimNameSpace Function

```
int pmTrimNameSpace(void)
```

If the current PMAPI context corresponds to a version 1 PCP archive log of performance metrics (as collected by `pmlogger` in PCP 1.x releases), and `pmLoadNameSpace` has been called to load a local PMNS, then this PMNS is trimmed to exclude metrics for which no description can be found in the archive. The PMNS is further trimmed to remove empty subtrees that contain no performance metrics.

Since the PCP archives usually contain some subset of all metrics named in a local PMNS, `pmTrimNameSpace` effectively trims the application's PMNS to contain only the names of the metrics in the archive. Before any trimming, the PMNS is restored to the state as of the completion of the last `pmLoadNameSpace` operation; so the effects of consecutive calls to `pmTrimNameSpace` with archive contexts are not cumulative.

If the current PMAPI context corresponds to a host, rather than an archive, the PMNS reverts to all names loaded into the PMNS at completion of the last `pmLoadNameSpace` operation. For example, any trimming is undone.

The PMNS services in PCP 2.x avoid the need for a local PMNS in most cases (and by default use only the PMNS of the metrics in a PCP archive) so that applications would typically not call `pmTrimNameSpace`.

#### 3.7.1.11 `pmUnloadNameSpace` Function

```
int pmUnloadNameSpace(void)
```

If a local PMNS was loaded with `pmLoadNameSpace`, calling `pmUnloadNameSpace` frees up the memory associated with the PMNS and force all subsequent Name Space functions to use the distributed PMNS. If `pmUnloadNameSpace` is called before calling `pmLoadNameSpace`, it has no effect.

### 3.7.2 PMAPI Metrics Description Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) metric description services.

#### 3.7.2.1 `pmLookupDesc` Function

```
int pmLookupDesc(pmID pmid, pmDesc *desc)
```

Given a Performance Metric Identifier (PMID) as *pmid*, `pmLookupDesc` returns the associated `pmDesc` structure through the parameter *desc* from the current PMAPI context. For more information about `pmDesc`, see Section 3.4, page 57.

#### 3.7.2.2 `pmLookupInDomText` Function

```
int pmLookupInDomText(pmInDom indom, int level, char **buffer)
```

Provided the source of metrics from the current PMAPI context is a host, retrieve descriptive text about the performance metrics instance domain identified by *indom*.

The *level* argument should be `PM_TEXT_ONELINE` for a one-line summary, or `PM_TEXT_HELP` for a more verbose description suited to a help dialogue. The space pointed to by *buffer* is allocated in `pmLookupInDomText` with `malloc`, and it is the responsibility of the caller to free unneeded space; see the `malloc(3C)` and `free(3C)` man pages.

The help text files used to implement `pmLookupInDomText` are created using `newhelp` and accessed by the appropriate PMDA response to requests forwarded to the PMDA by PMCD. Further details may be found in Section 2.4.4.

### 3.7.2.3 `pmLookupText` Function

```
int pmLookupText(pmID pmid, int level, char **buffer)
```

Provided the source of metrics from the current PMAPI context is a host, retrieve descriptive text about the performance metric identified by *pmid*. The argument *level* should be `PM_TEXT_ONELINE` for a one-line summary, or `PM_TEXT_HELP` for a more verbose description, suited to a help dialogue.

The space pointed to by *buffer* is allocated in `pmLookupText` with `malloc`, and it is the responsibility of the caller to `free` the space when it is no longer required; see the `malloc(3C)` and `free(3C)` man pages.

The help text files used to implement `pmLookupText` are created using `newhelp` and accessed by the appropriate PMDA in response to requests forwarded to the PMDA by PMCD. Further details may be found in Section 2.4.4.

## 3.7.3 PMAPI Instance Domain Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) instance domain services.

### 3.7.3.1 `pmGetInDom` Function

```
int pmGetInDom(pmInDom indom, int **instlist, char ***namelist)
```

In the current PMAPI context, locate the description of the instance domain *indom*, and return through *instlist* the internal instance identifiers for all instances, and through *namelist* the full external identifiers for all instances. The number of instances found is returned as the function value (or less than zero to indicate an error).

The resulting lists of instance identifiers (*instlist* and *namelist*), and the names that the elements of *namelist* point to, are allocated by `pmGetInDom` with two calls to `malloc`, and it is the responsibility of the caller to use `free(instlist)` and `free(namelist)` to release the space when it is no longer required. When the result of `pmGetInDom` is less than one, both *instlist* and *namelist* are undefined (no space is allocated, and so calling `free` is a bad idea); see the `malloc(3C)` and `free(3C)` man pages.

### 3.7.3.2 pmLookupInDom Function

```
int pmLookupInDom(pmInDom indom, char *name)
```

For the instance domain *indom*, in the current PMAPI context, locate the instance with the external identification given by *name*, and return the internal instance identifier.

### 3.7.3.3 pmNameInDom Function

```
int pmNameInDom(pmInDom indom, int inst, char **name)
```

For the instance domain *indom*, in the current PMAPI context, locate the instance with the internal instance identifier given by *inst*, and return the full external identification through *name*. The space for the value of *name* is allocated in `pmNameInDom` with `malloc`, and it is the responsibility of the caller to free the space when it is no longer required; see the `malloc(3C)` and `free(3C)` man pages.

## 3.7.4 PMAPI Context Services

Table 3-1 shows which of the three components of a PMAPI context (metrics source, instance profile, and collection time) are relevant for various PMAPI functions. Those PMAPI functions not shown in this table either manipulate the PMAPI context directly, or are executed independently of the current PMAPI context.

**Table 3-1** Context Components of PMAPI Functions

Function Name	Metrics Source	Instance Profile	Collection Time	Notes
<code>pmAddProfile</code>	Yes	Yes		
<code>pmDelProfile</code>	Yes	Yes		
<code>pmDupContext</code>	Yes	Yes	Yes	
<code>pmFetch</code>	Yes	Yes	Yes	
<code>pmFetchArchive</code>	Yes		Yes	(1)
<code>pmGetArchiveEnd</code>	Yes			(1)
<code>pmGetArchiveLabel</code>	Yes			(1)
<code>pmGetChildren</code>	Yes			(5)

Function Name	Metrics Source	Instance Profile	Collection Time	Notes
pmGetChildrenStatus	Yes			(5)
pmGetPMNSLocation	Yes			
pmGetInDom	Yes		Yes	(2)
pmGetInDomArchive	Yes			(1)
pmLookupDesc	Yes			(3)
pmLookupInDom	Yes		Yes	(2)
pmLookupInDomArchive	Yes			(1,2)
pmLookupInDomText	Yes			(4)
pmLookupName	Yes			(5)
pmLookupText	Yes			(4)
pmNameAll	Yes			(5)
pmNameID	Yes			(5)
pmNameInDom	Yes		Yes	(2)
pmNameInDomArchive	Yes			(1,2)
pmSetMode	Yes		Yes	
pmStore	Yes			(6)
pmTraversePMNS	Yes			(5)
pmTrimNameSpace	Yes			

Notes:

1. Operation supported only for PMAPI contexts where the source of metrics is an archive.
2. A specific instance domain is included in the arguments to these functions, and the result is independent of the instance profile for any PMAPI context.
3. The metadata that describes a performance metric is sensitive to the source of the metrics, but independent of any instance profile and of the collection time.
4. Operation is supported only for PMAPI contexts where the source of metrics is a host. The text associated with a metric is assumed to be invariant with time and is

definitely insensitive to the current members of the instance domain. In all cases this information is unavailable from an archive context (it is not included in the archive logs), and is directly available from a PMDA via PMCD in the other cases.

5. PMNS service functions using a local PMNS do **not** depend on the PMAPI context, whereas PCP 2.x distributed PMNS services are dependent on the source of metrics.
6. This operation is supported only for contexts where the source of the metrics is a host. Further, the instance identifiers are included in the argument to the function, and the effects upon the current values of the metrics are immediate (retrospective changes are not allowed). Consequently, from the current PMAPI context, neither the instance profile nor the collection time influence the result of this function.

#### 3.7.4.1 pmNewContext Function

```
int pmNewContext(int type, char *name)
```

The `pmNewContext` function may be used to establish a new PMAPI context. The source of metrics is identified by *name*, and may be a host name (*type* is `PM_CONTEXT_HOST`) or the basename of an archive log (*type* is `PM_CONTEXT_ARCHIVE`).

In the case where *type* is `PM_CONTEXT_LOCAL`, *name* is ignored, and the context uses a stand-alone connection to the PMDA methods used by PMCD. When this type of context is in effect, the range of accessible performance metrics is constrained to those from the operating system, and optionally the `proc` and `sample` PMDAs.

The initial instance profile is set up to select all instances in all instance domains, and the initial collection time is the current time at the time of each request for a host, or the time at the start of the log for an archive. In the case of archives, the initial collection time results in the earliest set of metrics being returned from the archive at the first `pmFetch`.

Once established, the association between a PMAPI context and a source of metrics is fixed for the life of the context; however, functions are provided to independently manipulate both the instance profile and the collection time components of a context.

The function returns a “handle” that may be used in subsequent calls to `pmUseContext`. This new PMAPI context stays in effect for all subsequent context sensitive calls across the PMAPI until another call to `pmNewContext` is made, or the context is explicitly changed with a call to `pmDupContext` or `pmUseContext`.

### 3.7.4.2 pmDestroyContext Function

```
int pmDestroyContext(int handle)
```

The PMAPI context identified by *handle* is destroyed. Typically, this implies terminating a connection to PMCD or closing an archive file, and orderly clean-up. The PMAPI context must have been previously created using `pmNewContext` or `pmDupContext`.

On success, `pmDestroyContext` returns zero. If *handle* was the current PMAPI context, then the current context becomes undefined. This means the application must explicitly re-establish a valid PMAPI context with `pmUseContext`, or create a new context with `pmNewContext` or `pmDupContext`, before the next PMAPI operation requiring a PMAPI context.

### 3.7.4.3 pmDupContext Function

```
int pmDupContext(void)
```

Replicate the current PMAPI context (source, instance profile, and collection time). This function returns a handle for the new context, which may be used with subsequent calls to `pmUseContext`. The newly replicated PMAPI context becomes the current context.

### 3.7.4.4 pmUseContext Function

```
int pmUseContext(int handle)
```

Calling `pmUseContext` causes the current PMAPI context to be set to the context identified by *handle*. The value of *handle* must be one returned from an earlier call to `pmNewContext` or `pmDupContext`.

Below the PMAPI, all contexts used by an application are saved in their most recently modified state, so `pmUseContext` restores the context to the state it was in the last time the context was used, not the state of the context when it was established.

### 3.7.4.5 pmWhichContext Function

```
int pmWhichContext(void)
```

Returns the handle for the current PMAPI context (source, instance profile, and collection time).



#### 3.7.4.6 pmAddProfile Function

```
int pmAddProfile(pmInDom indom, int numinst, int instlist[])
```

Add new instance specifications to the instance profile of the current PMAPI context. At its simplest, instances identified by the *instlist* argument for the *indom* instance domain are added to the instance profile. The list of instance identifiers contains *numinst* values.

If *indom* equals `PM_INDOM_NULL`, or *numinst* is zero, then all instance domains are selected. If *instlist* is `NULL`, then all instances are selected. To enable all available instances in all domains, use this syntax:

```
pmAddProfile(PM_INDOM_NULL, 0, NULL).
```

#### 3.7.4.7 pmDelProfile Function

```
int pmDelProfile(pmInDom indom, int numinst, int instlist[])
```

Delete instance specifications from the instance profile of the current PMAPI context. In the simplest variant, the list of instances identified by the *instlist* argument for the *indom* instance domain is removed from the instance profile. The list of instance identifiers contains *numinst* values.

If *indom* equals `PM_INDOM_NULL`, then all instance domains are selected for deletion. If *instlist* is `NULL`, then all instances in the selected domains are removed from the profile. To disable all available instances in all domains, use this syntax:

```
pmDelProfile(PM_INDOM_NULL, 0, NULL)
```

#### 3.7.4.8 pmSetMode Function

```
int pmSetMode(int mode, const struct timeval *when, int delta)
```

This function defines the collection time and mode for accessing performance metrics and metadata in the current PMAPI context. This mode affects the semantics of subsequent calls to the following PMAPI functions: `pmFetch`, `pmFetchArchive`, `pmLookupDesc`, `pmGetInDom`, `pmLookupInDom`, and `pmNameInDom`.

The `pmSetMode` function requires the current PMAPI context to be of type `PM_CONTEXT_ARCHIVE`.

The *when* parameter defines a time origin, and all requests for metadata (metrics descriptions and instance identifiers from the instance domains) are processed to

reflect the state of the metadata as of the time origin. For example, use the last state of this information at, or before, the time origin.

If the *mode* is `PM_MODE_INTERP` then, in the case of `pmFetch`, the underlying code uses an interpolation scheme to compute the values of the metrics from the values recorded for times in the proximity of the time origin.

If the *mode* is `PM_MODE_FORW`, then, in the case of `pmFetch`, the collection of recorded metric values is scanned forward, until values for at least one of the requested metrics is located after the time origin. Then all requested metrics stored in the PCP archive at that time are returned with a corresponding time stamp. This is the default mode when an archive context is first established with `pmNewContext`.

If the *mode* is `PM_MODE_BACK`, then the situation is the same as for `PM_MODE_FORW`, except a `pmFetch` is serviced by scanning the collection of recorded metrics backward for metrics before the time origin.

After each successful `pmFetch`, the time origin is reset to the time stamp returned through the `pmResult`.

The `pmSetMode` parameter *delta* defines an additional number of time unit that should be used to adjust the time origin (forward or backward) after the new time origin from the `pmResult` has been determined. This is useful when moving through archives with a mode of `PM_MODE_INTERP`. The high-order bits of the *mode* parameter field is also used to optionally set the units of time for the *delta* field. To specify the units of time, use the `PM_XTB_SET` macro with one of the values `PM_TIME_NSEC`, `PM_TIME_MSEC`, `PM_TIME_SEC`, or so on as follows:

```
PM_MODE_INTERP | PM_XTB_SET(PM_TIME_XXXX)
```

If no units are specified, the default is to interpret *delta* as milliseconds.

Using these mode options, an application can implement replay, playback, fast forward, or reverse for performance metric values held in a PCP archive log by alternating calls to `pmSetMode` and `pmFetch`.

In Example 3-8, the code fragment may be used to dump only those values stored in correct temporal sequence, for the specified performance metric `my.metric.name`:

**Example 3-8** Dumping Values in Temporal Sequence

```
int     sts;
pmID   pmid;
char   *name = ``my.metric.name``;
sts = pmNewContext(PM_CONTEXT_ARCHIVE, ``myarchive``);
sts = pmLookupName(1, &name, &pmid);
for ( ; ; ) {
    sts = pmFetch(1, &pmid, &result);
    if (sts < 0)
        break;
    /* dump value(s) from result->vset[0]->vlist[] */
    pmFreeResult(result);
}
```

Alternatively, the code fragment in Example 3-9 may be used to replay interpolated metrics from an archive in reverse chronological order, at ten-second intervals (of recorded time):

**Example 3-9** Replaying Interpolated Metrics

```
int     sts;
pmID   pmid;
char   *name = ``my.metric.name``;
struct timeval  endtime;
sts = pmNewContext(PM_CONTEXT_ARCHIVE, ``myarchive``);
sts = pmLookupName(1, &name, &pmid);
sts = pmGetArchiveEnd(&endtime);
sts = pmSetMode(PM_MODE_INTERP, &endtime, -10000);
while (pmFetch(1, &pmid, &result) != PM_ERR_EOL) {
    /*
     * process interpolated metric values as of result->timestamp
     */
    pmFreeResult(result);
}
```

### 3.7.4.9 pmReconnectContext Function

```
int pmReconnectContext(int handle)
```

As a result of network, host, or PMCD (Performance Metrics Collection Daemon) failure, an application's connection to PMCD may be established and then lost.

The function `pmReconnectContext` allows an application to request that the PMAPI context identified by *handle* be re-established, provided the associated PMCD is accessible.

---

**Note:** *handle* may or may not be the current context.

---

To avoid flooding the system with reconnect requests, `pmReconnectContext` attempts a reconnection only after a suitable delay from the previous attempt. This imposed restriction on the reconnect re-try time interval uses a default exponential back-off so that the initial delay is 5 seconds after the first unsuccessful attempt, then 10 seconds, then 20 seconds, then 40 seconds, and then 80 seconds thereafter. The intervals between reconnection attempts may be modified using the environment variable `PMCD_RECONNECT_TIMEOUT` and the time to wait before an attempted connection is deemed to have failed is controlled by the `PMCD_CONNECT_TIMEOUT` environment variable; see the `PCPIntro(1)` man page.

If the reconnection succeeds, `pmReconnectContext` returns *handle*. Note that even in the case of a successful reconnection, `pmReconnectContext` does not change the current PMAPI context.

### 3.7.4.10 pmGetContextHostName Function

```
const char *pmGetContextHostName(int id)
```

Given a valid PCP context identifier previously created with `pmNewContext` or `pmDupContext`, the `pmGetContextHostName` function provides a possibility to retrieve a host name associated with a context regardless of the context type.

If *id* is not a valid PCP context identifier, this function returns a zero length string and therefore never fails.

### 3.7.5 PMAPI Timezone Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) timezone services.

#### 3.7.5.1 `pmNewContextZone` Function

```
int pmNewContextZone(void)
```

If the current PMAPI context is an archive, the `pmNewContextZone` function uses the timezone from the archive label record to set the current reporting timezone. The current reporting timezone affects the timezone used by `pmCtime` and `pmLocaltime`.

If the current PMAPI context corresponds to a host source of metrics, `pmNewContextZone` executes a `pmFetch` to retrieve the value for the metric `pmcd.timezone` and uses that to set the current reporting timezone.

In both cases, the function returns a value to identify the current reporting timezone that may be used in a subsequent call to `pmUseZone` to restore this reporting timezone.

`PM_ERR_NOCONTEXT` indicates the current PMAPI context is not valid. A return value less than zero indicates a fatal error from a system call, most likely `malloc`.

#### 3.7.5.2 `pmNewZone` Function

```
int pmNewZone(const char *tz)
```

The `pmNewZone` function sets the current reporting timezone, and returns a value that may be used in a subsequent call to `pmUseZone` to restore this reporting timezone. The current reporting timezone affects the timezone used by `pmCtime` and `pmLocaltime`.

The `tz` argument defines a timezone string, in the format described for the `TZ` environment variable. See the `environ(5)` man page.

A return value less than zero indicates a fatal error from a system call, most likely `malloc`.

### 3.7.5.3 pmUseZone Function

```
int pmUseZone(const int tz_handle)
```

In the `pmUseZone` function, `tz_handle` identifies a reporting timezone as previously established by a call to `pmNewZone` or `pmNewContextZone`, and this becomes the current reporting timezone. The current reporting timezone effects the timezone used by `pmCtime` and `pmLocaltime`.

A return value less than zero indicates the value of `tz_handle` is not legal.

### 3.7.5.4 pmWhichZone Function

```
int pmWhichZone(char **tz)
```

The `pmWhichZone` function returns the handle of the current timezone, as previously established by a call to `pmNewZone` or `pmNewContextZone`. If the call is successful (that is, there exists a current reporting timezone), a non-negative integer is returned and `tz` is set to point to a static buffer containing the timezone string itself. The current reporting timezone effects the timezone used by `pmCtime` and `pmLocaltime`.

A return value less than zero indicates there is no current reporting timezone.

## 3.7.6 PMAPI Metrics Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) metrics services.

### 3.7.6.1 pmFetch Function

```
int pmFetch(int numpmid, pmID pmidlist[], pmResult **result)
```

The most common PMAPI operation is likely to be calls to `pmFetch`, specifying a list of PMIDs (for example, as constructed by `pmLookupName`) through `pmidlist` and `numpmid`. The call to `pmFetch` is executed in the context of a source of metrics, instance profile, and collection time, previously established by calls to the functions described in Section 3.7.4.

The principal result from `pmFetch` is returned as a tree structured `result`, described in the Section 3.5.

If one value (for example, associated with a particular instance) for a requested metric is unavailable at the requested time, then there is no associated `pmValue` structure in

the result. If there are no available values for a metric, then *numval* is zero and the associated `pmValue[]` instance is empty; *valfmt* is undefined in these circumstances, but *pmid* is correctly set to the PMID of the metric with no values.

If the source of the performance metrics is able to provide a reason why no values are available for a particular metric, this reason is encoded as a standard error code in the corresponding *numval*; see the `pmErr(1)` and `pmErrStr(3)` man pages. Since all error codes are negative, values for a requested metric are unavailable if *numval* is less than or equal to zero.

The argument definition and the result specifications have been constructed to ensure that for each PMID in the requested *pmidlist* there is exactly one `pmValueSet` in the result, and that the PMIDs appear in exactly the same sequence in both *pmidlist* and *result*. This makes the number and order of entries in *result* completely deterministic, and greatly simplifies the application programming logic after the call to `pmFetch`.

The result structure returned by `pmFetch` is dynamically allocated using one or more calls to `malloc` and specialized allocation strategies, and should be released when no longer required by calling `pmFreeResult`. Under no circumstances should `free` be called directly to release this space.

As common error conditions are encoded in the result data structure, only serious events (such as loss of connection to PMCD, `malloc` failure, and so on) would cause an error value to be returned by `pmFetch`. Otherwise, the value returned by the `pmFetch` function is zero.

In Example 3-10, the code fragment dumps the values (assumed to be stored in the *lval* element of the `pmValue` structure) of selected performance metrics once every 10 seconds:

**Example 3-10** PMAPI Metrics Services

```
int      numpmid, i, j, sts;
pmID     pmidlist[10];
pmResult *result;
time_t   now;
/* set up PMAPI context, numpmid and pmidlist[] ... */
while ((sts = pmFetch(&result)) >= 0) {
    now = (time_t)result->timestamp.tv_sec;
    printf("\n@ %s", ctime(&now));
    for (i = 0; i < result->numpmid; i++) {
        printf("PMID: %s", pmIDStr(result->vset[i]->pmid));
        for (j = 0; j < result->vset[i]->numval; j++) {
```

```
        printf(" 0x%x", result->vset[i]->vlist[j].value.lval);
        putchar('\n');
    }
}
pmFreeResult(result);
sleep(10);
}
```

---

**Note:** If a response is not received back from PMCD within 10 seconds, the `pmFetch` times out and returns `PM_ERR_TIMEOUT`. This is most likely to occur when the PMAPI client and PMCD are communicating over a slow network connection, but may also occur when one of the hosts is extremely busy. The time out period may be modified using the `PMCD_REQUEST_TIMEOUT` environment variable; see the `PCPIntro(1)` man page.

---

### 3.7.6.2 `pmFreeResult` Function

```
void pmFreeResult(pmResult *result)
```

Release the storage previously allocated for a result by `pmFetch`.

### 3.7.6.3 `pmStore` Function

```
int pmStore(const pmResult *request)
```

In some special cases it may be helpful to modify the current values of performance metrics in one or more underlying domains, for example to reset a counter to zero, or to modify a *metric*, which is a control variable within a Performance Metric Domain.

The `pmStore` function is a lightweight inverse of `pmFetch`. The caller must build the `pmResult` data structure (which could have been returned from an earlier `pmFetch` call) and then call `pmStore`. It is an error to pass a *request* to `pmStore` in which the `numval` field within any of the `pmValueSet` structure has a value less than one.

The current PMAPI context must be one with a host as the source of metrics, and the current value of the nominated metrics is changed. For example, `pmStore` cannot be used to make retrospective changes to information in a PCP archive log.



### 3.7.7 PMAPI Record-Mode Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) record-mode services.

#### 3.7.7.1 `pmRecordAddHost` Function

```
int pmRecordAddHost(const char *host, int isdefault, pmRecordHost **rhp)
```

The `pmRecordAddHost` function adds hosts once `pmRecordSetup` has established a new recording session. The `pmRecordAddHost` function along with the `pmRecordSetup` and `pmRecordControl` functions are used to create a PCP archive.

`pmRecordAddHost` is called for each host that is to be included in the recording session. A new `pmRecordHost` structure is returned via `rhp`. It is assumed that PMCD is running on the host as this is how `pmlogger` retrieves the required performance metrics.

If this host is the default host for the recording session, `isdefault` is nonzero. This ensures that the corresponding archive appears first in the PCP archive *folio*. Hence the tools used to replay the archive *folio* make the correct determination of the archive associated with the default host. At most one host per recording session may be nominated as the default host.

The calling application writes the desired `pmlogger` configuration onto the `stdio` stream returned via the `f_config` field in the `pmRecordHost` structure.

`pmRecordAddHost` returns 0 on success and a value less than 0 suitable for decoding with `pmErrStr` on failure. The value `EINVAL` has the same interpretation as `errno` being set to `EINVA`.

#### 3.7.7.2 `pmRecordControl` Function

```
int pmRecordControl(pmRecordHost *rhp, int request, const char *options)
```

Arguments may be optionally added to the command line that is used to launch `pmlogger` by calling the `pmRecordControl` function with a request of `PM_REC_SETARG`. The `pmRecordControl` along with the `pmRecordSetup` and `pmRecordAddHost` functions are used to create a PCP archive.

The argument is passed via `options` and one call to `pmRecordControl` is required for each distinct argument. An argument may be added for a particular `pmlogger` instance identified by `rhp`. If the `rhp` argument is `NULL`, the argument is added for all `pmlogger` instances that are launched in the current recording session.

Independent of any calls to `pmRecordControl` with a request of `PM_REC_SETARG`, each `pmlogger` instance is automatically launched with the following arguments: `-c`, `-h`, `-l`, `-x`, and the basename for the PCP archive log.

To commence the recording session, call `pmRecordControl` with a request of `PM_REC_ON`, and `rhp` must be `NULL`. This launches one `pmlogger` process for each host in the recording session and initializes the `fd_ipc`, `logfile`, `pid`, and `status` fields in the associated `pmRecordHost` structure(s).

To terminate a `pmlogger` instance identified by `rhp`, call `pmRecordControl` with a request of `PM_REC_OFF`. If the `rhp` argument to `pmRecordControl` is `NULL`, the termination request is broadcast to all `pmlogger` processes in the current recording session. An informative dialogue is generated directly by each `pmlogger` process.

To display the current status of the `pmlogger` instance identified by `rhp`, call `pmRecordControl` with a request of `PM_REC_STATUS`. If the `rhp` argument to `pmRecordControl` is `NULL`, the status request is broadcast to all `pmlogger` processes in the current recording session. The display is generated directly by each `pmlogger` process.

To detach a `pmlogger` instance identified by `rhp`, allow it to continue independent of the application that launched the recording session and call `pmRecordControl` with a request of `PM_REC_DETACH`. If the `rhp` argument to `pmRecordControl` is `NULL`, the detach request is broadcast to all `pmlogger` processes in the current recording session.

`pmRecordControl` returns 0 on success and a value less than 0 suitable for decoding with `pmErrStr` on failure. The value `EINVAL` has the same interpretation as `errno` being set to `EINVA`.

`pmRecordControl` returns `PM_ERR_IPC` if the associated `pmlogger` process has already exited.

### 3.7.7.3 `pmRecordSetup` Function

```
FILE *pmRecordSetup(const char *folio, const char *creator, int replay)
```

The `pmRecordSetup` function along with the `pmRecordAddHost` and `pmRecordControl` functions may be used to create a PCP archive on the fly to support record-mode services for PMAPI client applications.

Each record mode session involves one or more PCP archive logs; each is created using a dedicated `pmlogger` process, with an overall Archive Folio format as

understood by the `pmaf` command, to name and collect all of the archive logs associated with a single recording session.

The `pmRecordHost` structure is used to maintain state information between the creator of the recording session and the associated `pmlogger` process(es). The structure, shown in Example 3-11, is defined as:

**Example 3-11** `pmRecordHost` Structure

```
typedef struct {
    FILE *f_config; /* caller writes pmlogger configuration here */
    int fd_ipc; /* IPC channel to pmlogger */
    char *logfile; /* full pathname for pmlogger error logfile */
    pid_t pid; /* process id for pmlogger */
    int status; /* exit status, -1 if unknown */
} pmRecordHost;
```

In Procedure 3-1, the functions are used in combination to create a recording session.

**Procedure 3-1** Creating a Recording Session

1. Call `pmRecordSetup` to establish a new recording session. A new Archive Folio is created using the name *folio*. If the *folio* file or directory already exists, or if it cannot be created, this is an error. The application that is creating the session is identified by creator (most often this would be the same as the global PMAPI application name, `pmProgname`). If the application knows how to create its own configuration file to replay the recorded session, `replay` should be nonzero. The `pmRecordSetup` function returns a stdio stream onto which the application writes the text of any required replay configuration file.
2. For each host that is to be included in the recording session, call `pmRecordAddHost`. A new `pmRecordHost` structure is returned via *rhp*. It is assumed that PMCD is running on the host as this is how `pmlogger` retrieves the required performance metrics. See Section 3.7.7.1 for more information.
3. Optionally, add arguments to the command line that is used to launch `pmlogger` by calling `pmRecordControl` with a request of `PM_REC_SETARG`. The argument is passed via options and one call to `pmRecordControl` is required for each distinct argument. See Section 3.7.7.2 for more information.
4. To commence the recording session, call `pmRecordControl` with a request of `PM_REC_ON`, and *rhp* must be NULL.

5. To terminate a `pmlogger` instance identified by `rhp`, call `pmRecordControl` with a request of `PM_REC_OFF`.
6. To display the current status of the `pmlogger` instance identified by, `rhp`, call `pmRecordControl` with a request of `PM_REC_STATUS`.
7. To detach a `pmlogger` instance identified by `rhp`, allow it to continue independent of the application that launched the recording session, call `pmRecordControl` with a request of `PM_REC_DETACH`.

The calling application should not close any of the returned `stdio` streams; `pmRecordControl` performs this task when recording is commenced.

Once `pmlogger` has been started for a recording session, `pmlogger` assumes responsibility for any dialogue with the user in the event that the application that launched the recording session should exit, particularly without terminating the recording session.

By default, information and dialogues from `pmlogger` is displayed using `xconfirm`. This default is based on the assumption that most applications launching a recording session are GUI-based. In the event that `xconfirm` fails to display the information (for example, because the `DISPLAY` environment variable is not set), `pmlogger` writes on its own `stderr` stream (not the `stderr` stream of the launching process). The output is assigned to the `XXXXXX.host.log` file. For convenience, the full pathname to this file is provided via the `logfile` field in the `pmRecordHost` structure.

If the `options` argument to `pmRecordControl` is not `NULL`, this string may be used to pass additional arguments to `xconfirm` in those cases where a dialogue is to be displayed. One use of this capability is to provide a `-geometry` string to control the placement of the dialogue.

Premature termination of a launched `pmlogger` process may be determined using the `pmRecordHost` structure, by calling `select` on the `fd_ipc` field or polling the `status` field that will contain the termination status from `waitpid` if known, or `-1`.

These functions create a number of files in the same directory as the `folio` file named in the call to `pmRecordSetup`. In all cases, the `XXXXXX` component is the result of calling `mktemp`.

- If `replay` is nonzero, `XXXXXX` is the creator's replay configuration file, else an empty control file, used to guarantee uniqueness.
- The `folio` file is the PCP Archive Folio, suitable for use with the `pmaf` command.

- The `XXXXXX.host.config` file is the `pmlogger` configuration for each host. If the same host is used in different calls to `pmRecordAddHost` within the same recording session, one of the letters 'a' through 'z' is appended to the `XXXXXX` part of all associated file names to ensure uniqueness.
- `XXXXXX.host.log` is `stdout` and `stderr` for the `pmlogger` instance for each host.
- The `XXXXXX.host.{0,meta,index}` files comprise a single PCP archive for each host.

`pmRecordSetup` may return `NULL` in the event of an error. Check `errno` for the real cause. The value `EINVAL` typically means that the order of calls to these functions is not correct; that is, there is an obvious state associated with the current recording session that is maintained across calls to the functions.

For example, calling `pmRecordControl` before calling `pmRecordAddHost` at least once, or calling `pmRecordAddHost` before calling `pmRecordSetup` would produce an `EINVAL` error.

### 3.7.8 PMAPI Archive-Specific Services

The functions described in this section provide archive-specific services.

#### 3.7.8.1 `pmGetArchiveLabel` Function

```
int pmGetArchiveLabel(int handle, pmLogLabel *lp)
```

Provided the current PMAPI context is associated with a PCP archive log, the `pmGetArchiveLabel` function may be used to fetch the label record from the archive. The structure returned through `lp` is as shown in Example 3-12:

#### Example 3-12 `pmLogLabel` Structure

```
/*
 * Label Record at the start of every log file - as exported above the PMAPI ...
 */
#define PM_TZ_MAXLEN          40
#define PM_LOG_MAXHOSTLEN    64
#define PM_LOG_MAGIC         0x50052600
#define PM_LOG_VERS01        0x1
#define PM_LOG_VERS02        0x2
#define PM_LOG_VOL_TI        -2      /* temporal index */
#define PM_LOG_VOL_META      -1      /* meta data */
```

```
typedef struct {
    int ll_magic; /* PM_LOG_MAGIC | log format version no. */
    pid_t ll_pid; /* PID of logger */
    struct timeval ll_start; /* start of this log */
    char ll_hostname[PM_LOG_MAXHOSTLEN]; /* name of collection host */
    char ll_tz[PM_TZ_MAXLEN]; /* $TZ at collection host */
} pmLogLabel;
```

### 3.7.8.2 pmGetArchiveEnd Function

```
int pmGetArchiveEnd(struct timeval *tvp)
```

Provided the current PMAPI context is associated with a PCP archive log, `pmGetArchiveEnd` finds the logical end of file (after the last complete record in the archive), and returns the last recorded time stamp with *tvp*. This time stamp may be passed to `pmSetMode` to reliably position the context at the last valid log record, for example, in preparation for subsequent reading in reverse chronological order.

For archive logs that are not concurrently being written, the physical end of file and the logical end of file are co-incident. However, if an archive log is being written by `pmlogger` at the same time that an application is trying to read the archive, the logical end of file may be before the physical end of file due to write buffering that is not aligned with the logical record boundaries.

### 3.7.8.3 pmGetInDomArchive Function

```
int pmGetInDomArchive(pmInDom indom, int **instlist, char ***namelist )
```

Provided the current PMAPI context is associated with a PCP archive log, `pmGetInDomArchive` scans the metadata to generate the union of all instances for the instance domain *indom* that can be found in the archive log, and returns through *instlist* the internal instance identifiers, and through *namelist* the full external identifiers.

This function is a specialized version of the more general PMAPI function `pmGetInDom`.

The function returns the number of instances found (a value less than zero indicates an error).

The resulting lists of instance identifiers (*instlist* and *namelist*), and the names that the elements of *namelist* point to, are allocated by `pmGetInDomArchive` with two calls to `malloc`, and it is the responsibility of the caller to use `free(instlist)` and

`free(namelist)` to release the space when it is no longer required; see the `malloc(3C)` and `free(3C)` man pages.

When the result of `pmGetInDomArchive` is less than one, both `instlist` and `namelist` are undefined (no space is allocated; so calling `free` is a singularly bad idea).

#### 3.7.8.4 `pmLookupInDomArchive` Function

```
int pmLookupInDomArchive(pmInDom indom, const char *name)
```

Provided the current PMAPI context is associated with a PCP archive log, `pmLookupInDomArchive` scans the metadata for the instance domain `indom`, locates the first instance with the external identification given by `name`, and returns the internal instance identifier.

This function is a specialized version of the more general PMAPI function `pmLookupInDom`.

The `pmLookupInDomArchive` function returns a positive instance identifier on success.

#### 3.7.8.5 `pmNameInDomArchive` Function

```
int pmNameInDomArchive(pmInDom indom, int inst, char **name)
```

Provided the current PMAPI context is associated with a PCP archive log, `pmNameInDomArchive` scans the metadata for the instance domain `indom`, locates the first instance with the internal instance identifier given by `inst`, and returns the full external instance identification through `name`. This function is a specialized version of the more general PMAPI function `pmNameInDom`.

The space for the value of `name` is allocated in `pmNameInDomArchive` with `malloc`, and it is the responsibility of the caller to free the space when it is no longer required; see the `malloc(3C)` and `free(3C)` man pages.

#### 3.7.8.6 `pmFetchArchive` Function

```
int pmFetchArchive(pmResult **result)
```

This is a variant of `pmFetch` that may be used only when the current PMAPI context is associated with a PCP archive log. The `result` is instantiated with all of the metrics (and instances) from the next archive record; consequently, there is no notion of a list of desired metrics, and the instance profile is ignored.

It is expected that `pmFetchArchive` would be used to create utilities that scan archive logs (for example, `pmDumpLog`), and the more common access to the archives would be through the `pmFetch` interface.

### 3.7.9 PMAPI Time Control Services

The PMAPI provides a common framework for client applications to control time and to synchronize time with other applications. The user interface component of this service is fully described in the companion *Performance Co-Pilot User's and Administrator's Guide*. See also the `pmtime(1)` man page.

This service is most useful when processing PCP archive logs, to control parameters such as the current archive position, update interval, replay rate, and timezone, but it can also be used in live mode to control a subset of these parameters. Applications such as `pmchart`, `pmview`, `oview`, and `pmval` use the time control services to connect to an instance of the time control server process, `pmtime`, which provides a uniform graphical user interface to the time control services.

A full description of the PMAPI time control functions along with code examples can be found in man pages as listed in Table 3-2:

**Table 3-2** Time Control Functions in PMAPI

Man Page	Synopsis of Time Control Function
<code>pmCtime(3)</code>	Formats the date and time for a reporting timezone.
<code>pmLocaltime(3)</code>	Converts the date and time for a reporting timezone.
<code>pmParseTimeWindow(3)</code>	Parses time window command line arguments.
<code>pmTimeConnect(3)</code>	Connects to a time control server via a command socket.
<code>pmTimeDisconnect(3)</code>	Closes the command socket to the time control server.
<code>pmTimeGetPort(3)</code>	Obtains the port name of the current time control server.
<code>pmTimeRecv(3)</code>	Blocks until the time control server sends a command message.
<code>pmTimeSendAck(3)</code>	Acknowledges completion of the step command.
<code>pmTimeSendBounds(3)</code>	Specifies beginning and end of archive time period.
<code>pmTimeSendMode(3)</code>	Requests time control server to change to a new VCR mode.



---

Man Page	Synopsis of Time Control Function
<code>pmTimeSendPosition(3)</code>	Requests time control server to change position or update intervals.
<code>pmTimeSendTimezone(3)</code>	Requests time control server to change timezones.
<code>pmTimeShowDialog(3)</code>	Changes the visibility of the time control dialogue.
<code>pmTimeGetStatePixmap(3)</code>	Returns array of pixmaps representing supplied time control state.

---

## 3.8 PMAPI Ancillary Support Services

The functions described in this section provide services that are complementary to, but not necessarily a part of, the distributed manipulation of performance metrics delivered by the PCP components.

### 3.8.1 `pmGetConfig` Function

```
char *pmGetConfig(const char *variable)
```

The `pmGetConfig` function searches for a variable first in the environment and then, if one is not found, in the PCP configuration file and returns the string result. If a variable is not already in the environment, it is added with a call to the `putenv` function before returning.

The default location of the PCP configuration file is `/etc/pcp.conf`, but this location may be changed by setting `PCP_CONF` in the environment to a new location, as described in the `pcp_conf(4)` man page.

If the variable is not found in either the environment or the PCP configuration file (or the PCP configuration file is not found and `PCP_CONF` is not set in the environment), then a fatal error message is printed and the process will exit. Although this sounds drastic, it is the only course of action available because the PCP configuration or installation is fatally flawed.

If this function returns, the returned value points to a string in the environment; and so although the function returns the same type as the `getenv` function (which should probably be a `const char *`), changing the content of the string is not recommended.

### 3.8.2 pmErrStr Function

```
char *pmErrStr(int code)
```

This function translates an error code into a text string, suitable for generating a diagnostic message. By convention within PCP, all error codes are negative. The small values are assumed to be negated versions of the UNIX error codes as defined in `errno.h`, and the strings returned are according to `strerror`. The large, negative error codes are PMAPI error conditions, and `pmErrStr` returns an appropriate PMAPI error string, as determined by `code`.

The string value is held in a single static buffer; so the returned value is valid only until the next call to `pmErrStr`.

### 3.8.3 pmExtractValue Function

```
int pmExtractValue(int valfmt, const pmValue *ival, int itype,
pmAtomValue *oval, int otype)
```

The `pmValue` structure is embedded within the `pmResult` structure, which is used to return one or more performance metrics; see the `pmFetch` man page.

All performance metric values may be encoded in a `pmAtomValue` union, defined in Example 3-13:

#### Example 3-13 pmAtomValue Structure

```
/* Generic Union for Value-Type conversions */
typedef union {
    _int32_t    l;        /* 32-bit signed */
    _uint32_t   ul;       /* 32-bit unsigned */
    _int64_t    ll;       /* 64-bit signed */
    _uint64_t   ull;      /* 64-bit unsigned */
    float       f;        /* 32-bit floating point */
    double      d;        /* 64-bit floating point */
    char        *cp;     /* char ptr */
    void        *vp;     /* void ptr */
} pmAtomValue;
```

The `pmExtractValue` function provides a convenient mechanism for extracting values from the `pmValue` part of a `pmResult` structure, optionally converting the data type, and making the result available to the application programmer.

The *itype* argument defines the data type of the input value held in *ival* according to the storage format defined by *valfmt* (see the `pmFetch` man page). The *otype* argument defines the data type of the result to be placed in *oval*. The value for *itype* is typically extracted from a `pmDesc` structure, following a call to `pmLookupDesc` for a particular performance metric.

Table 3-3 defines the various possibilities for the type conversion. The input type (*itype*) is shown vertically, and the output type (*otype*) horizontally. The following rules apply:

- Y means the conversion is always acceptable.
- N means conversion can never be performed (function returns `PM_ERR_CONV`).
- P means the conversion may lose accuracy (but no error status is returned).
- T means the result may be subject to high-order truncation (if this occurs the function returns `PM_ERR_TRUNC`).
- S means the conversion may be impossible due to the sign of the input value (if this occurs the function returns `PM_ERR_SIGN`).

If an error occurs, *oval* is set to zero (or NULL).

---

**Note:** Note that some of the conversions involving the `PM_TYPE_STRING` and `PM_TYPE_AGGREGATE` types are indeed possible, but are marked N; the rationale is that `pmExtractValue` should not attempt to duplicate functionality already available in the C library through `sscanf` and `sprintf`.

---

**Table 3-3** PMAPI Type Conversion

TYPE	STRIN							AGGR
	32	U32	64	U64	FLOAT	DBLE	G	
32	Y	S	Y	S	P	P	N	N
U32	T	Y	Y	Y	P	P	N	N
64	T	T,S	Y	S	P	P	N	N
u64	T	T	T	Y	P	P	N	N
FLOAT	P, T	P, T, S	P, T	P, T, S	Y	Y	N	N

TYPE	32	U32	64	U64	FLOAT	DBLE	STRIN G	AGGR
DBLE	P, T	P, T, S	P, T	P, T, S	P	Y	N	N
STRING	N	N	N	N	N	N	Y	N
AGGR	N	N	N	N	N	N	N	Y

In the cases where multiple conversion errors could occur, the first encountered error is returned, and the order of checking is not defined.

If the output conversion is to one of the pointer types, such as *otype* `PM_TYPE_STRING` or `PM_TYPE_AGGREGATE`, then the value buffer is allocated by `pmExtractValue` using `malloc`, and it is the caller's responsibility to free the space when it is no longer required; see the `malloc(3C)` and `free(3C)` man pages.

Although this function appears rather complex, it has been constructed to assist the development of performance tools that convert values, whose type is known only through the `type` field in a `pmDesc` structure, into a canonical type for local processing.

### 3.8.4 `pmConvScale` Function

```
int
pmConvScale(int type, const pmAtomValue *ival, const pmUnits *iunit,
pmAtomValue *oval, pmUnits *ounit)
```

Given a performance metric value pointed to by *ival*, multiply it by a scale factor and return the value in *oval*. The scaling takes place from the units defined by *iunit* into the units defined by *ounit*. Both input and output units must have the same dimensionality.

The performance metric type for both input and output values is determined by *type*, the value for which is typically extracted from a `pmDesc` structure, following a call to `pmLookupDesc` for a particular performance metric.

`pmConvScale` is most useful when values returned through `pmFetch` (and possibly extracted using `pmExtractValue`) need to be normalized into some canonical scale and units for the purposes of computation.

### 3.8.5 pmUnitsStr Function

```
const char *pmUnitsStr(const pmUnits *pu)
```

As an aid to labeling graphs and tables, or for error messages, `pmUnitsStr` takes a dimension and scale specification as per `pu`, and returns the corresponding text string.

`pu` is typically from a `pmDesc` structure, for example, as returned by `pmLookupDesc`.

If `*pu` were `{1, -2, 0, PM_SPACE_MBYTE, PM_TIME_MSEC, 0}`, then the result string would be `Mbyte/sec^2`.

The string value is held in a single static buffer; so concurrent calls to `pmUnitsStr` may not produce the desired results.

### 3.8.6 pmIDStr Function

```
const char *pmIDStr(pmID pmid)
```

For use in error and diagnostic messages, return a human readable version of the specified PMID, with each of the internal domain, cluster, and item subfields appearing as decimal numbers, separated by periods.

The string value is held in a single static buffer; so concurrent calls to `pmIDStr` may not produce the desired results.

### 3.8.7 pmInDomStr Function

```
const char *pmInDomStr(pmInDom indom)
```

For use in error and diagnostic messages, return a human readable version of the specified instance domain identifier, with each of the internal domain and serial subfields appearing as decimal numbers, separated by periods.

The string value is held in a single static buffer; so concurrent calls to `pmInDomStr` may not produce the desired results.

### 3.8.8 pmTypeStr Function

```
const char *pmTypeStr(int type)
```

Given a performance metric type, produce a terse ASCII equivalent, appropriate for use in error and diagnostic messages.

Examples are “32” (for `PM_TYPE_32`), “U64” (for `PM_TYPE_U64`), “AGGREGATE” (for `PM_TYPE_AGGREGATE`), and so on.

The string value is held in a single static buffer; so concurrent calls to `pmTypeStr` may not produce the desired results.

### 3.8.9 `pmAtomStr` Function

```
const char *pmAtomStr(const pmAtomValue *avp, int type)
```

Given the `pmAtomValue` identified by *avp*, and a performance metric *type*, generate the corresponding metric value as a string, suitable for diagnostic or report output.

The string value is held in a single static buffer; so concurrent calls to `pmAtomStr` may not produce the desired results.

### 3.8.10 `pmNumberStr` Function

```
const char *pmNumberStr(double value)
```

The `pmNumberStr` function returns the address of a static 8-byte buffer that holds a null-byte terminated representation of value suitable for output with fixed-width fields.

The value is scaled using multipliers in powers of one thousand (the decimal kilo) and has a bias that provides greater precision for positive numbers as opposed to negative numbers. The format depends on the sign and magnitude of *value*.

### 3.8.11 `pmPrintValue` Function

```
void pmPrintValue(FILE *f, int valfmt, int type, const pmValue *val,  
int minwidth)
```

*type*The value of a single performance metric (as identified by *val*) is printed on the standard I/O stream identified by *f*. The value of the performance metric is interpreted according to the format of *val* as defined by *valfmt* (from a `pmValueSet` within a `pmResult`) and the generic description of the metric’s type from a `pmDesc` structure, passed in through.

If the converted value is less than *minwidth* characters wide, it will have leading spaces to pad the output to a width of *minwidth* characters.

Example 3-14 illustrates using `pmPrintValue` to print the values from a `pmResult` structure returned via `pmFetch`:

**Example 3-14** Using `pmPrintValue` to Print Values

```

int          numpmid, i, j, sts;
pmID        pmidlist[10];
pmDesc      desc[10];
pmResult    *result;
/* set up PMAPI context, numpmid and pmidlist[] ... */
/* get metric descriptors */
for (i = 0; i < numpmid; i++) {
    if ((sts = pmLookupDesc(pmidlist[i], &desc[i])) < 0) {
        printf("pmLookupDesc(pmid=%s): %s\n",
               pmIDStr(pmidlist[i]), pmErrStr(sts));
        exit(1);
    }
}
if ((sts = pmFetch(numpmid, pmidlist, &result)) >= 0) {
    /* once per metric */
    for (i = 0; i < result->numpmid; i++) {
        printf("PMID: %s", pmIDStr(result->vset[i]->pmid));
        /* once per instance for this metric */
        for (j = 0; j < result->vset[i]->numval; j++) {
            printf(" [%d]", result->vset[i]->vlist[j].inst);
            pmPrintValue(stdout, result->vset[i]->valfmt,
                        desc[i].type,
                        &result->vset[i]->vlist[j],
                        8);
        }
        putchar('\n');
    }
    pmFreeResult(result);
}
else
    printf("pmFetch: %s\n", pmErrStr(sts));

```

### 3.8.12 pmflush Function

```
int pmflush(void);
```

The `pmflush` function causes the internal buffer which is shared with `pmprintf` to be either displayed in a window, printed on standard error, or flushed to a file and the internal buffer to be cleared.

The `PCP_STDERR` environment variable controls the output technique used by `pmflush`:

- If `PCP_STDERR` is unset, the text is written onto the `stderr` stream of the caller.
- If `PCP_STDERR` is set to the literal reserved word `DISPLAY`, then the text is displayed as a GUI dialog using `xconfirm`.

The `pmflush` function returns a value of zero on successful completion. A negative value is returned if an error was encountered, and this can be passed to `pmErrStr` to obtain the associated error message.

### 3.8.13 pmprintf Function

```
int pmprintf(const char *fmt, ... /*args*/);
```

The `pmprintf` function appends the formatted message string to an internal buffer shared by the `pmprintf` and `pmflush` functions, without actually producing any output. The `fmt` argument is used to control the conversion, formatting, and printing of the variable length `args` list.

The `pmprintf` function uses the `tempnam` function to create a temporary file, using the value of the global variable `pmPrognam` as a prefix. This temporary file is deleted when `pmflush` is called.

On successful completion, `pmprintf` returns the number of characters transmitted. A negative value is returned if an error was encountered, and this can be passed to `pmErrStr` to obtain the associated error message.

### 3.8.14 pmSortInstances Function

```
void pmSortInstances(pmResult *result)
```

The `pmSortInstances` function may be used to guarantee that for each performance metric in the result from `pmFetch`, the instances are in ascending



internal instance identifier sequence. This is useful when trying to compute rates from two consecutive `pmFetch` results, where the underlying instance domain or metric availability is not static.

### 3.8.15 `pmParseInterval` Function

```
int pmParseInterval(const char *string, struct timeval *rslt,
char **errmsg)
```

The `pmParseInterval` function parses the argument string specifying an interval of time and fills in the `tv_sec` and `tv_usec` components of the `rslt` structure to represent that interval. The input string is most commonly the argument following a `-t` command line option to a PCP application, and the syntax is fully described in the `PCPIntro(1)` man page.

`pmParseInterval` returns 0 and `errmsg` is undefined if the parsing is successful. If the given string does not conform to the required syntax, the function returns -1 and a dynamically allocated error message string in `errmsg`.

The error message is terminated with a newline and includes the text of the input string along with an indicator of the position at which the error was detected as shown in the following example:

```
4minutes 30mumble
           ^ -- unexpected value
```

In the case of an error, the caller is responsible for calling `free` to release the space allocated for `errmsg`.

### 3.8.16 `pmParseMetricSpec` Function

```
int pmParseMetricSpec(const char *string, int isarch, char *source,
pmMetricSpec **rslt, char **errmsg)
```

The `pmParseMetricSpec` function accepts a `string` specifying the name of a PCP performance metric, and optionally the source (either a hostname or a PCP archive log filename) and instances for that metric. The syntax is described in the `PCPIntro(1)` man page.

If neither host nor archive component of the metric specification is provided, the `isarch` and `source` arguments are used to fill in the returned `pmMetricSpec`

structure. In Example 3-15, the `pmMetricSpec` structure, which is returned via `rsntp`, represents the parsed string.

**Example 3-15** `pmMetricSpec` Structure

```
typedef struct {
    int    isarch;      /* source type: 0 -> host, 1 -> archive */
    char   *source;     /* name of source host or archive */
    char   *metric;     /* name of metric */
    int    ninst;       /* number of instances, 0 -> all */
    char   *inst[1];    /* array of instance names */
} pmMetricSpec;
```

The `pmParseMetricSpec` function returns 0 if the given string was successfully parsed. In this case, all the storage allocated by `pmParseMetricSpec` can be released by a single call to the `free` function by using the address returned from `pmMetricSpec` via `rsntp`. The convenience macro `pmFreeMetricSpec` is a thinly disguised wrapper for `free`.

The `pmParseMetricSpec` function returns 0 if the given string was successfully parsed. It returns `PM_ERR_GENERIC` and a dynamically allocated error message string in `errmsg` if the given string does not parse. In this situation, the error message string can be released with the `free` function.

In the case of an error, `rsntp` is undefined. In the case of success, `errmsg` is undefined. If `rsntp->ninst` is 0, then `rsntp->inst[0]` is undefined.

## 3.9 PMAPI Programming Issues and Examples

The following issues and examples are provided to enable you to create better custom performance monitoring tools.

The source code for a sample client (`pmclient`) using the PMAPI is shipped as part of the `pcp.sw.demo` subsystem of the PCP product. See the `pmclient(1)` man page, and the source code, located in `/var/pcp/demos/pmclient`.

### 3.9.1 Symbolic Association between a Metric's Name and Value

A common problem in building specific performance tools is how to maintain the association between a performance metric's name, its access (instantiation) method, and the application program variable that contains the metric's value. Generally this results in code that is easily broken by bug fixes or changes in the underlying data structures. The PMAPI provides a uniform method for instantiating and accessing the values independent of the underlying implementation, although it does not solve the name-variable association problem. However, it does provide a framework within which a manageable solution may be developed.

Fundamentally, the goal is to be able to name a metric and reference the metric's value in a manner that is independent of the order of operations on other metrics; for example, to associate the `BINGO` macro with the name `sys.statistic.bingo`, and then be able to use `BINGO` to get at the value of the corresponding metric.

The one-to-one association between the ordinal position of the metric names is input to `pmLookupName` and the PMIDs returned by this function, and the one-to-one association between the PMIDs input to `pmFetch` and the values returned by this function provide the basis for an automated solution.

The tool `pmgenmap` takes the specification of a list of metric names and symbolic tags, in the order they should be passed to `pmLookupName` and `pmFetch`. For example:

```
# one line comment
mystuff {
    sys.statistic.bingo    BINGO
    sys.statistic.cache.miss  MISSED
}
```

This `pmgenmap` input produces the C code, in Example 3-16. It is suitable for including with the `#include` statement:

**Example 3-16** C Code Produced by `pmgenmap` Input

```
/*
 * Performance Metrics Name Space Map
 * Built by pmgenmap from the file
 * /usr/people/kenmcd/swa/ptg/src/kstat.pcp/x
 * on Thu Feb 24 20:37:53 EST 1994
 *
 * Do not edit this file!
 */
/* one line comment */
char *mystuff[] = {
#define BINGO 0
    "sys.statistic.bingo",
#define MISSED 1
    "sys.statistic.cache.miss",
};
```

### 3.9.2 Initializing New Metrics

Using the code generated by `pmgenmap`, you are now able to easily initialize the application's metric specifications as shown in Example 3-17:

**Example 3-17** Initializing Metric Specifications

```
#define MAX_MID 3
int      trip = 0;
int      numpmid = sizeof(mystuff)/sizeof(mystuff[0]);
double   duration;
pmResult *resp;
pmResult *prev;
pmID     pmidlist[MAX_MID];
pmLookupName(numpmid, mystuff, pmidlist);
```

At this stage, `pmidlist` contains the PMID for the two metrics of interest.

### 3.9.3 Iterative Processing of Values

Assuming the tool is required to report values every five seconds, use code similar to that in Example 3-18:

**Example 3-18** Iterative Processing

```
while (1) {
    pmFetch(numpmid, pmidlist, &resp);
    if (trip) {
        /* see pmclient.c for tv_sub() declaration */
        duration = tv_sub(&resp->timestamp, &prev->timestamp);
        /*
         * sys.boring.bozo is an instantaneous value,
         * so report the most recent value
         * sys.statistic.cache.miss is a free running counter,
         * so report the rate over the last two samples
         */
        printf("%6d %5.2f\n", resp->vset[BOZO]->vlist[0].value.lval,
            (resp->vset[MISSED]->vlist[0].value.lval -
             prev->vset[MISSED]->vlist[0].value.lval) / duration);
    }
    if (trip >= 1)
        pmFreeResult(prev);
    else
        trip++;
    prev = resp;
    sleep(5);
}
```

### 3.9.4 Accommodating Program Evolution

The flexibility provided by the PMAPI and the `pmgenmap` utility is demonstrated by Example 3-19. Consider the requirement for reporting a third metric `sys.boring.new` (an instantaneous value) in the middle of the two already reported. This example shows how to add the line to the middle of the specification file:

**Example 3-19** Adding a Metric

```
sys.boring.new NEW
```

Then regenerate the #include file, and amend the printf statement as shown in

```
printf("%6d %6d %5.2f\n",
    resp->vlist[BOZO]->vlist[0].value.lval,
    resp->vlist[NEW]->vlist[0].value.lval,
    (resp->vlist[MISSED]->vlist[0].value.lval -
     prev->vlist[MISSED]->vlist[0].value.lval) / duration);
```

### 3.9.5 Handling PMAPI Errors

In Example 3-20, the simple but complete PMAPI application demonstrates the recommended style for handling PMAPI error conditions.

#### Example 3-20 PMAPI Error Handling

```
#include <stdio.h>
#include <pcp/pmapi.h>
int
main(int argc, char* argv[])
{
    int                sts = 0;
    char               *host = "localhost";
    char               *metric = "mem.freemem";
    pmID               pmid;
    pmDesc             desc;
    pmResult           *result;

    sts = pmNewContext(PM_CONTEXT_HOST, host);
    if (sts < 0) {
        fprintf(stderr, "Error connecting to pmcd on %s: %s\n",
            host, pmErrStr(sts));
        exit(1);
    }
    sts = pmLookupName(1, &metric, &pmid);
    if (sts < 0) {
        fprintf(stderr, "Error looking up %s: %s\n", metric,
            pmErrStr(sts));
        exit(1);
    }
    sts = pmLookupDesc(pmid, &desc);
    if (sts < 0) {
        fprintf(stderr, "Error getting descriptor for %s:%s: %s\n",
```

```

        host, metric, pmErrStr(sts));
    exit(1);
}
sts = pmFetch(1, &pmid, &result);
if (sts < 0) {
    fprintf(stderr, "Error fetching %s:%s: %s\n", host, metric,
            pmErrStr(sts));
    exit(1);
}
sts = result->vset[0]->numval;
if (sts < 0) {
    fprintf(stderr, "Error fetching %s:%s: %s\n", host, metric,
            pmErrStr(sts));
    exit(1);
}
fprintf(stdout, "%s:%s = ", host, metric);
if (sts == 0)
    puts("(no value)");
else {
    pmValueSet      *vsp = result->vset[0];
    pmPrintValue(stdout, vsp->valfmt, desc.type,
                &vsp->vlist[0], 5);
    printf(" %s\n", pmUnitsStr(&desc.units));
}
return 0;
}

```

### 3.9.6 Compiling and Linking PMAPI Applications

Typical PMAPI applications require the following line to include the function prototype and data structure definitions used by the PMAPI.

```
#include <pcp/pmapi.h>
```

Some applications may also require these header files: `pcp/impl.h` and `pcp/pmda.h`.

The run-time environment of the PMAPI is mostly found in `libpcp.so`; so to link a generic PMAPI application requires something akin to the following command:

```
cc mycode.c -lpcp
```





## Trace PMDA

This chapter provides an introduction to the design of the trace Performance Metrics Domain Agent (PMDA), in an effort to explain how to configure the agent optimally for a particular problem domain. This information supplements the functional coverage which the man pages provide to both the agent and the library interfaces.

The chapter also includes information on how to use the trace PMDA and the associated library (`libpcp_trace`) for instrumenting applications. The example programs are installed in `/var/pcp/demos/trace` from the `pcp.sw.trace` subsystem.

### 4.1 Performance Instrumentation and Tracing

The `pcp_trace` library provides function calls for identifying sections of a program as transactions or events for examination by the trace PMDA, a user command called `pmdatrace`. The `pcp_trace` library is described in the `pmdatrace(3)` man page

The monitoring of transactions using the Performance Co-Pilot (PCP) infrastructure begins with a `pmtracebegin` call. Time is recorded from there to the corresponding `pmtraceend` call (with matching tag identifier). A transaction in progress can be cancelled by calling `pmtraceabort`.

A second form of program instrumentation is available with the `pmtracepoint` function. This is a simpler form of monitoring that exports only the number of times a particular point in a program is passed. The `pmtraceobs` and `pmtracecount` functions have similar semantics, but allows an arbitrary numeric value to be passed to the trace PMDA.

The `pmdatrace` command is a PMDA that exports transaction performance metrics from application processes using the `pcp_trace` library; see the `pmdatrace(1)` man page for details.

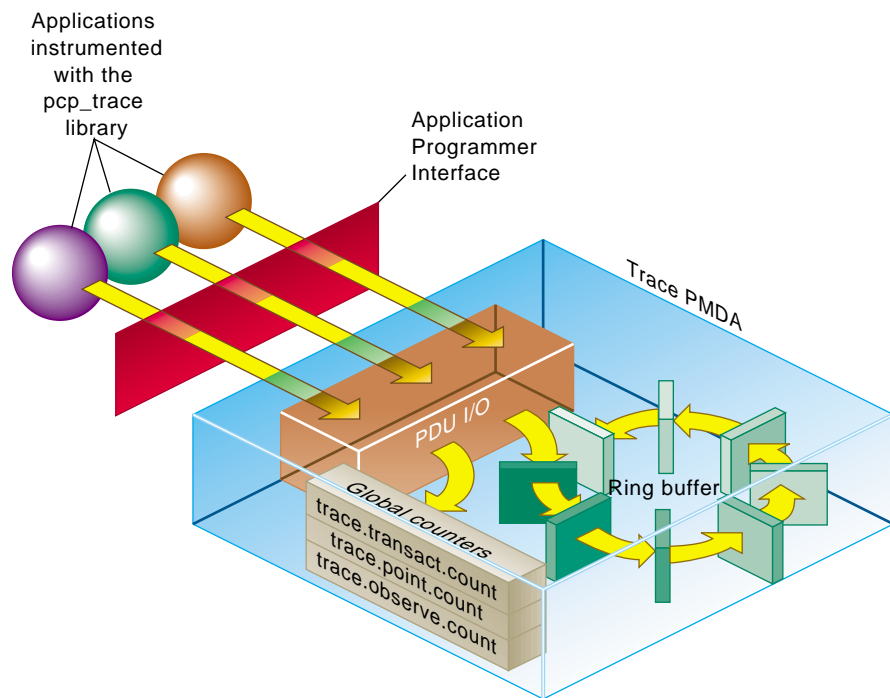
For a complete introduction to performance tracing, refer to the Web-based *PCP Tutorial*, which contains the `trace.html` file covering this topic.

## 4.2 Trace PMDA Design

Trace PMDA design covers application interaction, sampling techniques, and configuring the trace PMDA.

### 4.2.1 Application Interaction

Figure 4-1 describes the general state maintained within the trace PMDA.



**Figure 4-1** Trace PMDA Overview

Applications that are linked with the `libpcp_trace` library make calls through the trace Application Programming Interface (API). These calls result in interprocess communication of trace data between the application and the trace PMDA. This data consists of an identification tag and the performance data associated with that

particular tag. The trace PMDA aggregates the incoming information and periodically updates the exported summary information to describe activity in the recent past.

As each protocol data unit (PDU) is received, its data is stored in the current working buffer. At the same time, the global counter associated with the particular tag contained within the PDU is incremented. The working buffer contains all performance data that has arrived since the previous time interval elapsed. For additional information about the working buffer, see Section 4.2.2.2.

## 4.2.2 Sampling Techniques

The trace PMDA employs a rolling-window periodic sampling technique. The arrival time of the data at the trace PMDA in conjunction with the length of the sampling period being maintained by the PMDA determines the recency of the data exported by the PMDA. Through the use of rolling-window sampling, the trace PMDA is able to present a more accurate representation of the available trace data at any given time than it could through use of simple periodic sampling.

The rolling-window sampling technique affects the metrics in Example 4-1:

### Example 4-1 Rolling-Window Sampling Technique

```
trace.observe.rate  
trace.counter.rate  
trace.point.rate  
trace.transact.ave_time  
trace.transact.max_time  
trace.transact.min_time  
trace.transact.rate
```

The remaining metrics are either global counters, control metrics, or the last seen observation value. Section 4.3, page 114, documents in more detail all metrics exported by the trace PMDA.

### 4.2.2.1 Simple Periodic Sampling

The simple periodic sampling technique uses a single historical buffer to store the history of events that have occurred over the sampling interval. As events occur, they are recorded in the working buffer. At the end of each sampling interval, the working buffer (which at that time holds the historical data for the sampling interval just finished) is copied into the historical buffer, and the working buffer is cleared. It is ready to hold new events from the sampling interval now starting.

#### 4.2.2.2 Rolling-Window Periodic Sampling

In contrast to simple periodic sampling with its single historical buffer, the rolling-window periodic sampling technique maintains a number of separate buffers. One buffer is marked as the current working buffer, and the remainder of the buffers hold historical data. As each event occurs, the current working buffer is updated to reflect it.

At a specified interval, the current working buffer and the accumulated data that it holds is moved into the set of historical buffers, and a new working buffer is used. The specified interval is a function of the number of historical buffers maintained.

The primary advantage of the rolling-window sampling technique is seen at the point where data is actually exported. At this point, the data has a higher probability of reflecting a more recent sampling period than the data exported using simple periodic sampling.

The data collected over each sample duration and exported using the rolling-window sampling technique provides a more up-to-date representation of the activity during the most recently completed sample duration than simple periodic sampling as shown in Figure 4-2.

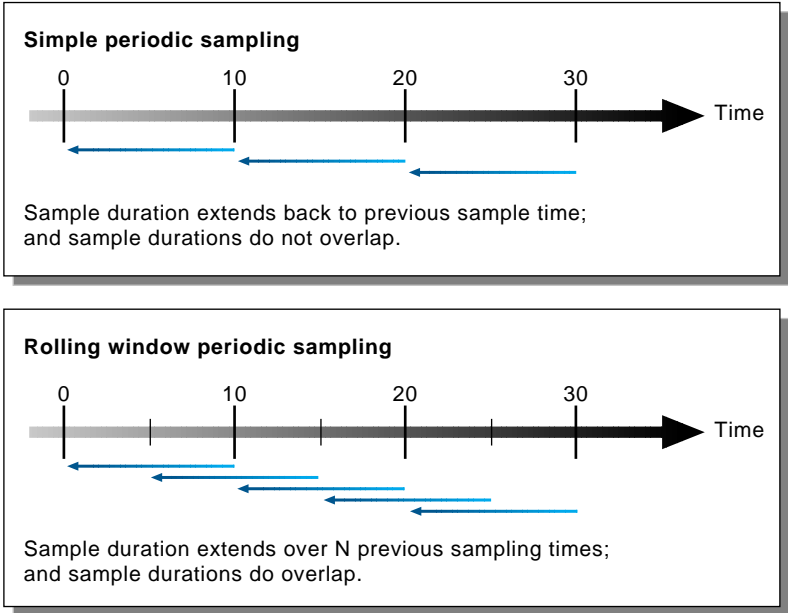


Figure 4-2 Sample Duration Comparison

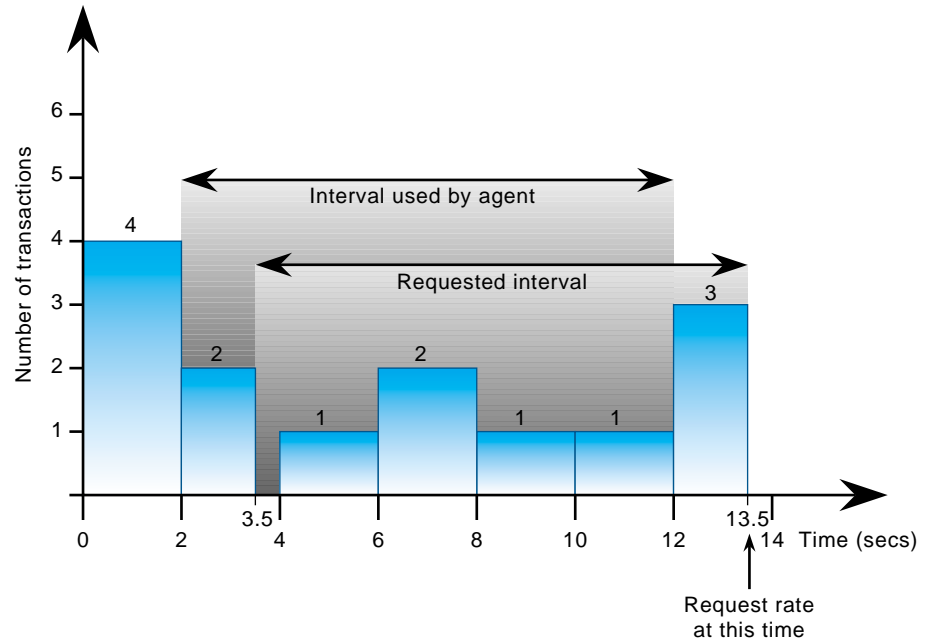
The trace PMDA allows the length of the sample duration to be configured, as well as the number of historical buffers that are maintained. The rolling-window approach is implemented in the trace PMDA as a ring buffer (see Figure 4-1).

When the current working buffer is moved into the set of historical buffers, the least recent historical buffer is cleared of data and becomes the new working buffer.

4.2.2.3 Rolling-Window Periodic Sampling Example

Consider the scenario where you want to know the rate of transactions over the last 10 seconds. You set the sampling rate for the trace PMDA to 10 seconds and fetch the metric `trace.transact.rate`. So if in the last 10 seconds, 8 transactions took place, the transaction rate would be 8/10 or 0.8 transactions per second.

The trace PMDA does not actually do this. It instead does its calculations automatically at a subinterval of the sampling interval. Reconsider the 10-second scenario. It has a calculation subinterval of 2 seconds as shown in Figure 4-3.



**Figure 4-3** Sampling Intervals

If at 13.5 seconds, you request the transaction rate, you receive a value of 0.7 transactions per second. In actual fact, the transaction rate was 0.8, but the trace PMDA did its calculations on the sampling interval from 2 seconds to 12 seconds, and not from 3.5 seconds to 13.5 seconds. For efficiency, the trace PMDA calculates the metrics on the last 10 seconds every 2 seconds. As a result, the PMDA is not driven each time a fetch request is received to do a calculation.

### 4.2.3 Configuring the Trace PMDA

The trace PMDA is configurable primarily through command-line options. The list of command-line options in Table 4-1 is not exhaustive, but it identifies those options which are particularly relevant to tuning the manner in which performance data is collected.

**Table 4-1** Selected Command-Line Options

Option	Description
Access controls	The trace PMDA offers host-based access control. This control allows and disallows connections from instrumented applications running on specified hosts or groups of hosts. Limits to the number of connections allowed from individual hosts can also be mandated.
Sample duration	The interval over which metrics are to be maintained before being discarded is called the sample duration.
Number of historical buffers	The data maintained for the sample duration is held in a number of internal buffers within the trace PMDA. These are referred to as historical buffers. This number is configurable so that the rolling window effect can be tuned within the sample duration.

Option	Description
Counter and observation metric units	Since the data being exported by the <code>trace.observe.value</code> and <code>trace.counter.count</code> metrics are user-defined, the trace PMDA by default exports these metrics with a type of "none." A framework is provided that allows the user to make the type more specific (for example, bytes per second) and allows the exported values to be plotted along with other performance metrics of similar units by tools like <code>pmchart</code> .
Instance domain refresh	The set of instances exported for each of the trace metrics can be cleared through the storable <code>trace.control.reset</code> metric.

### 4.3 Trace API

The `libpcp_trace` Application Programming Interface (API) is called from C, C++, Fortran, and Java. Each language has access to the complete set of functionality offered by `libpcp_trace`. In some cases, the calling conventions differ slightly between languages. This section presents an overview of each of the different tracing mechanisms offered by the API, as well as an explanation of their mappings to the actual performance metrics exported by the trace PMDA.



### 4.3.1 Transactions

Paired calls to the `pmtracebegin` and `pmtraceend` API functions result in transaction data being sent to the trace PMDA with a measure of the time interval between the two calls. This interval is the transaction service time. Using the `pmtraceabort` call causes data for that particular transaction to be discarded. The trace PMDA exports transaction data through the following `trace.transact` metrics listed in Table 4-2:

**Table 4-2** `trace.transact` Metrics

Metric	Description
<code>trace.transact.ave_time</code>	The average service time per transaction type. This time is calculated over the last sample duration.
<code>trace.transact.count</code>	The running count for each transaction type seen since the trace PMDA started.
<code>trace.transact.max_time</code>	The maximum service time per transaction type within the last sample duration.
<code>trace.transact.min_time</code>	The minimum service time per transaction type within the last sample duration.
<code>trace.transact.rate</code>	The average rate at which each transaction type is completed. The rate is calculated over the last sample duration.
<code>trace.transact.total_time</code>	The cumulative time spent processing each transaction since the trace PMDA started running.

### 4.3.2 Point Tracing

Point tracing allows the application programmer to export metrics related to salient events. The `pmtracepoint` function is most useful when start and end points are not well defined. For example, this function is useful when the code branches in such a way that a transaction cannot be clearly identified, or when processing does not follow a transactional model, or when the desired instrumentation is akin to event rates rather than event service times. This data is exported through the `trace.point` metrics listed in Table 4-3:

**Table 4-3** `trace.point` Metrics

Metric	Description
<code>trace.point.count</code>	Running count of point observations for each tag seen since the trace PMDA started.
<code>trace.point.rate</code>	The average rate at which observation points occur for each tag within the last sample duration.

### 4.3.3 Observations and Counters

The `pmtraceobs` and `pmtracecount` functions have similar semantics to `pmtracepoint`, but also allow an arbitrary numeric value to be passed to the trace PMDA. The most recent value for each tag is then immediately available from the PMDA. Observation data is exported through the `trace.observe` metrics listed in Table 4-4:

**Table 4-4** `trace.observe` Metrics

Metric	Description
<code>trace.observe.count</code>	Running count of observations seen since the trace PMDA started.
<code>trace.observe.rate</code>	The average rate at which observations for each tag occur. This rate is calculated over the last sample duration.
<code>trace.observe.value</code>	The numeric value associated with the observation last seen by the trace PMDA.
<code>trace.counter</code>	Counter data is exported through the <code>trace.counter</code> metrics. The only difference between <code>trace.counter</code> and <code>trace.observe</code> metrics is that the numeric value of <code>trace.counter</code> must be a monotonic increasing count.

### 4.3.4 Configuring the Trace Library

The trace library is configurable through the use of environment variables listed in Table 4-5 as well as through the state flags listed in Table 4-6. Both provide diagnostic output and enable or disable the configurable functionality within the library.

**Table 4-5** Environment Variables

Name	Description
PCP_TRACE_HOST	The name of the host where the trace PMDA is running.
PCP_TRACE_PORT	TCP/IP port number on which the trace PMDA is accepting client connections.
PCP_TRACE_TIMEOUT	The number of seconds to wait until assuming that the initial connection is not going to be made, and timeout will occur. The default is three seconds.
PCP_TRACE_REQTIMEOUT	The number of seconds to allow before timing out on awaiting acknowledgment from the trace PMDA after trace data has been sent to it. This variable has no effect in the asynchronous trace protocol (refer to Table 4-6).
PCP_TRACE_RECONNECT	A list of values which represents the backoff approach that the <code>libpcp_trace</code> library routines take when attempting to reconnect to the trace PMDA after a connection has been lost. The list of values should be a positive number of seconds for the application to delay before making the next reconnection attempt. When the final value in the list is reached, that value is used for all subsequent reconnection attempts.

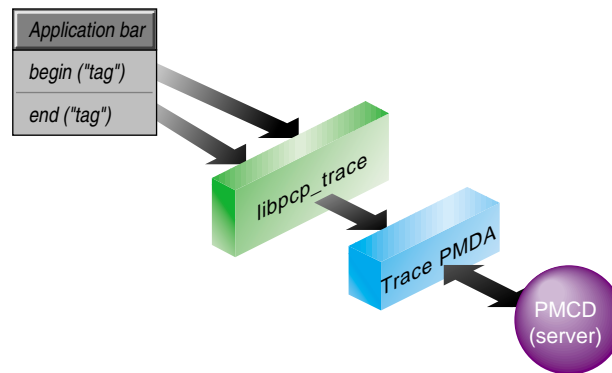
The Table 4-6 are used to customize the operation of the `libpcp_trace` routines. These are registered through the `pmtracestate` call, and they can be set either individually or together.

**Table 4-6** State Flags

Flag	Description
<code>PMTRACE_STATE_NONE</code>	The default. No state flags have been set, the fault-tolerant, synchronous protocol is used for communicating with the trace PMDA, and no diagnostic messages are displayed by the <code>libpcp_trace</code> routines.
<code>PMTRACE_STATE_API</code>	High-level diagnostics. This flag simply displays entry into each of the API routines.
<code>PMTRACE_STATE_COMMS</code>	Diagnostic messages related to establishing and maintaining the communication channel between application and PMDA.
<code>PMTRACE_STATE_PDU</code>	The low-level details of the trace protocol data units (PDU) is displayed as each PDU is transmitted or received.
<code>PMTRACE_STATE_PDUBUF</code>	The full contents of the PDU buffers are dumped as PDUs are transmitted and received.
<code>PMTRACE_STATE_NOAGENT</code>	Interprocess communication control. If this flag is set, it causes interprocess communication between the instrumented application and the trace PMDA to be skipped. This flag is a debugging aid for applications using <code>libpcp_trace</code> .
<code>PMTRACE_STATE_ASYNC</code>	Asynchronous trace protocol. This flag enables the asynchronous trace protocol so that the application does not block awaiting acknowledgment PDUs from the trace PMDA. In order for the flag to be effective, it must be set before using the other <code>libpcp_trace</code> entry points.

## 4.4 Instrumenting Applications to Export Performance Data

The relationship between an application, the `libpcp_trace` library, the trace PMDA and the rest of the PCP infrastructure is shown in Figure 4-4:



**Figure 4-4** Application and PCP Relationship

The `libpcp_trace` library is designed to encourage application developers (independent software vendors and end-user customers) to embed calls in their code that enable application performance data to be exported. When combined with system-level performance data, this feature allows total performance and resource demands of an application to be correlated with application activity.

For example, developers can provide the following application performance metrics:

- Computation state (especially for codes with major shifts in resource demands between phases of their execution)
- Problem size and parameters, that is, degree of parallelism throughput in terms of subproblems solved, iteration count, transactions, data sets inspected, and so on
- Service time by operation type

The `libpcp_trace` library approach offers a number of attractive features:

- A simple API for inserting instrumentation calls into an application as shown in the following example:

```
pmtracebegin('pass 1');
...
```

```
pmtraceend(``pass 1``);  
...  
pmtraceobs(``threads``, N);
```

- Trace routines that are called from C, C++, Fortran, and Java, and that are well suited to macro encapsulation for compile-time inclusion and exclusion.
- Shipped source code for a stub version of the library that enables the following:
  - Replacement by private debugging or development versions
  - Flexibility based on not being locked into a SGI program
  - Added functionality on SGI platforms, when the PCP version of the library is present
- A PCP version of the library that allows numerical observations, measures time between matching begin-end calls, and so on to be shipped to a PCP agent and then exported into the PCP infrastructure. As exporting is controlled by environment variables, the overhead is very low if the metrics are not being exported.

Once the application performance metrics are exported into the PCP framework, all of the PCP tools may be leveraged to provide performance monitoring and management, including:

- Two- and three-dimensional visualization of resource demands and performance, showing concurrent system activity and application activity.
- Transport of performance data over the network for distributed performance management.
- Archive logging for historical records of performance, most useful for problem diagnosis, postmortem analysis, performance regression testing, capacity planning, and benchmarking.
- Automated alarms when bad performance is observed. These apply both in real-time or when scanning archives.
- A toolkit approach that encourages customization. For example, a complete PCP for XYZ package could be offered for performance monitoring of application XYZ on SGI platforms.





## Acronyms

Table A-1 provides a glossary of the acronyms used in the Performance Co-Pilot (PCP) documentation, help cards, man pages, and user interface.

**Table A-1** Performance Co-Pilot Acronyms and Their Meanings

Acronym	Meaning
API	Application Programming Interface
DBMS	Database Management System
DNS	Domain Name Service
DSO	Dynamic Shared Object
I/O	Input/Output
IPC	Interprocess Communication
PCP	Performance Co-Pilot
PDU	Protocol data unit
PMAPI	Performance Metrics Application Programming Interface
PMCD	Performance Metrics Collection Daemon
PMCS	Performance Metrics Collection Subsystem
PMDA	Performance Metrics Domain Agent
PMID	Performance Metric Identifier
PMNS	Performance Metrics Name Space
TCP/IP	Transmission Control Protocol/Internet Protocol



---

## Index

### A

- access controls, 113
- acronyms, 123
- ancillary support services, 91
- Application Programming Interface, 53, 108, 114
- applications
  - compiling , 105
  - instrumentation, 120
  - interaction, 108
- architecture, 1, 12
- archive logs
  - context services, 72
  - performance data, 53, 56
  - performance management, 121
  - pmGetArchiveEnd function, 88
  - pmGetInDomArchive function, 88
  - retrospective sources, 6
  - time control services, 90
- archive-specific services, 87
- Array PMDA, 3
- arrays
  - instance description, 24
  - N dimensional data, 22
  - performance metrics, 61, 63
- asynchronous trace protocol, 118, 119
- audience, 1
- automated alarms, 121

### C

- caching PMDA, 16, 27
- Cisco PMDA, 3, 16
- client development, 8
- clusters, 5
- collection time, 56, 73, 74

- collection tools, 1
- collector hosts, 4
- COLOR\_INDOM, 25
- compiling and linking, 105
- computation state, 120
- configuration, 52, 118
- context services, 71
- control threads, 27
- counter semantics, 22
- customers, 120
- customization, 1, 121

### D

- daemon process method, 8
- data export, 120
- data structures, 19, 23
- dbpmda man page, 12, 45, 46
- dbx man page, 45
- debugging and testing, 45, 119
- debugging flags
  - See "flags", 45
- delays, 27
- design requirements, 11
- diagnostic output, 118
- dimensionality and scale, 58
- discrete semantics, 22
- distributed performance management
  - data transportation, 121
  - metrics collection, 3
- dkvis man page, 52
- dlopen man page, 7, 13
- DNS, 123
- domains
  - definition, 16
  - fields, 5

- numbers, 18
- dometric function, 68
- DSO, 123
  - architecture, 12
  - disadvantages, 15
  - implementation, 13
  - interface, 32
  - PMDA building, 7
  - PMDA initialization, 41
- dynamic shared object
  - See "DSO", 7

## E

- embedded calls, 120
- environment variables, 118, 121
- error handling, 104
- examples
  - alarm tools, 12
  - Install script, 51
  - programming issues, 101
  - Remove script, 52
  - rolling-window sampling, 111
  - simple and trivial PMDAs, 16
  - time control functions , 90
  - trace PMDA, 107
  - visualization tools, 12
- execv system call, 15
- exporting data, 26

## F

- flags
  - debugging, 45
  - state, 119
- fork system call, 15

## G

- glossary, 123

## H

- handle context, 78
- help text
  - creation, 51
  - initialization, 42
  - location, 47
  - PDU\_TEXT\_REQ, 13
  - pmLookupInDomText function, 69
  - pmLookupText function, 31, 70
  - structure specification, 12
  - terse and extended descriptions, 29
- historical buffers, 109, 111, 113

## I

- identification tags, 108
- implementation, 11
- independent software vendors, 120
- indom instance domain, 69, 75, 88
- information extraction, 26
- initialization, 102
- instance domain refresh, 114
- instance domain services, 70
- instantaneous semantics, 22
- instlist argument, 70, 75
- instrumentation, 107, 120
- integrating a PMDA, 47
- internal instance identifier, 60
- interpolated metrics, 77
- interprocess communication
  - PMTRACE\_STATE\_NOAGENT flag, 119
  - See "IPC", 7
- IPC
  - DSO, 7

- PMDA, 11
  - trace API, 108
- item numbers, 5
- iterative processing, 103

## L

- languages, 121
- latency, 27
- leaf node, 68
- libpcp\_trace library
  - Application Programming Interface, 114
  - entry points, 119
  - functions, 119
  - instrumenting applications, 107
- library reentrancy, 9

## M

- Makefile, 47
- metric description services, 69
- metrics
  - API, 53
  - definition, 16, 18
  - name and value, 101
- metrics and instances, 17
- metrics description services, 69
- metrics services, 80
- monitoring tools, 1
- mpvis man page, 52
- multidimensional arrays, 22
- multiple threads, 9

## N

- name space, 5, 28
- new metrics, 30, 102
- new PMDA, 51
- newhelp man page, 29

- NOW\_INDOM, 26

## O

- observation metric units, 114

## P

- parallelism, 120
- PCP
  - acronym, 123
  - description, 1
- PCP\_TRACE\_HOST variable, 118
- PCP\_TRACE\_PORT variable, 118
- PCP\_TRACE\_RECONNECT variable, 118
- PCP\_TRACE\_REQTIMEOUT variable, 118
- PCP\_TRACE\_TIMEOUT variable, 118
- PDU, 12, 109, 119, 123
- PDU\_DESC\_REQ, 13
- PDU\_FETCH, 13, 42
- PDU\_INSTANCE\_REQ, 13
- PDU\_PROFILE, 13
- PDU\_RESULT, 13, 42
- PDU\_TEXT\_REQ, 13
- performance instrumentation, 1, 107
- Performance Metric Identifier
  - See "PMID", 5
- performance metrics
  - See "metrics", 53
- Performance Metrics Application Programming Interface
  - See "PMAPI", 1
- Performance Metrics Collection Daemon
  - See "PMCD", 2
- Performance Metrics Collection Subsystem
  - See also "PMCS", 8
- Performance Metrics Domain Agent
  - See "PMDA", 1
- Performance Metrics Name Space

- See "PMNS", 5
- periodic sampling, 109
- pipe, 15
- PM\_CONTEXT\_ARCHIVE type, 73
- PM\_CONTEXT\_HOST type, 73
- PM\_ERR\_CONV error code, 31, 93
- PM\_ERR\_INST error code, 37
- PM\_ERR\_PPID error code, 31, 38
- PM\_ERR\_SIGN error code, 93
- PM\_ERR\_TIMEOUT error code, 82
- PM\_ERR\_TRUNC error code, 93
- PM\_IN\_NULL instance identifier, 55
- PM\_INDOM\_NULL instance domain
  - data structures, 21, 26
  - description, 55
  - pmAddProfile function, 75
  - pmDelProfile function, 75
- PM\_SEM\_COUNTER semantic type, 21
- PM\_SEM\_DISCRETE semantic type, 21
- PM\_SEM\_INSTANT semantic type, 21
- PM\_TYPE\_AGGREGATE type, 57
- PM\_TYPE\_NOSUPPORT value, 31, 58
- PM\_TYPE\_STRING type, 57, 93
- PM\_VAL\_INSITU value, 61
- pmAddProfile function, 13, 71, 75
- PMAPI
  - acronym, 123
  - ancillary support services, 91
  - application compiling, 105
  - archive-specific services, 87
  - client development, 8
  - context services, 71
  - current context, 56
  - description, 53
  - description services, 69
  - error handling, 63, 104
  - identifying metrics, 53
  - initializing new metrics, 102
  - instance domain services, 70
  - introduction, 1
  - iterative processing, 103
  - man page, 4
  - metrics services, 80
  - Name Space services, 64
  - program evolution, 103
  - programming issues, 101
  - programming style, 62
  - record-mode services, 83
  - See also "metrics", 54
  - time control services, 90
  - timezone services, 79
  - variable length arguments, 63
- pmAtomStr function, 31, 96
- pmAtomValue structure, 34
- PMCD
  - acronym, 123
  - distributed collection, 4
  - overview, 2
  - pmReconnectContext function, 78
- PMCD\_RECONNECT\_TIMEOUT variable, 78
- PMCD\_REQUEST\_TIMEOUT variable, 82
- pmchart command, 1, 12, 52, 114
- pmConvScale function, 31, 94
- PMCS, 8, 123
- PMDA
  - acronym, 123
  - architecture, 12
  - checklist, 11
  - development, 6
  - evolution, 30
  - help text, 29
  - initialization, 41
  - Install script, 47, 51
  - integration, 47
  - introduction, 1
  - man page, 4
  - removal, 51
  - structures, 39
  - trace, 107
  - writing, 11
- PMDA\_PPID macro, 20
- pmdacisco man page, 16
- pmdaConnect man page, 41, 43

- pmdaDaemon man page, 41, 43
- pmdaDesc callback, 32
- pmdaDSO man page, 41
- pmdaExt structure, 32, 40
- pmdaFetch callback, 32, 42
- pmdaGetOpt man page, 41, 43, 44
- pmdaInDom structure, 24
- pmdaInit man page, 24, 41, 42
- pmdaInstance callback, 32
- pmdaInstid structure, 23
- pmdaInterface structure, 39, 41
- pmdaMain man page, 43
- pmdaMetric structure, 20
- pmdaOpenLog man page, 43
- pmdaProfile callback, 32
- pmdaStore callback, 32, 36
- pmdaText callback, 32
- pmdatrace man page, 107
- pmdbg man page, 45
- pmDelProfile function, 71, 75
- pmDesc structure, 19, 31, 57, 58
- pmDestroyContext function, 74
- pmDupContext function, 71, 74
- pmErrStr function, 92
- pmExtractValue function, 31, 92, 94
- pmFetch function, 59, 61, 63, 71, 73, 75, 80–82, 89, 96, 98, 101
- pmFetch man page, 13, 31
- pmFetchArchive function, 71, 75, 89
- pmflush function, 98
- pmFreeResult function, 63, 81, 82
- pmgadgets command, 12
- pmGetArchiveEnd function, 71, 88
- pmGetArchiveLabel function, 71, 87
- pmGetChildren function, 63–65, 71
- pmGetChildrenStatus function, 72
- pmGetInDom function, 13, 63, 70, 72, 75, 88
- pmGetInDomArchive function, 72, 88
- pmGetPMNSLocation function, 65, 72
- PMID
  - acronym, 123
  - introduction, 5
- \_pmID\_int structure, 20
- pmIDStr function, 95
- pmie command, 12, 52
- pmieconf command, 12, 52
- \_pmInDom\_int structure, 24
- pmInDomStr function, 95
- pmLoadNameSpace function, 66
- pmlogger command, 12, 52
- pmlogger\_daily command, 52
- pmLookupDesc function, 13, 19, 31, 69, 72, 75, 93, 94
- pmLookupInDom function, 71, 72, 75
- pmLookupInDomArchive function, 72, 89
- pmLookupInDomText function, 69, 72
- pmLookupName function, 67, 72, 101
- pmLookupText function, 13, 31, 63, 70, 72
- pmNameAll function, 67
- pmNameID function, 63, 68, 72
- pmNameInDom function, 63, 71, 72, 75
- pmNameInDomArchive function, 72, 89
- pmNewContext function, 73
- pmNewContextZone function, 79
- pmNewZone function, 79
- PMNS
  - acronym, 123
  - distributed, 5
  - upgrade, 51
- pmns man page, 28
- pmNumberStr function, 96
- pmParseInterval function, 99
- pmParseMetricSpec function, 99
- pmprintf function, 98
- pmPrintValue function, 31, 96
- pmReconnectContext function, 78
- pmRecordAddHost function, 83
- pmRecordControl function, 83
- pmRecordSetup function, 84
- pmSetMode function, 72, 75, 88
- pmSortInstances function, 98
- pmStore function, 31, 59, 72, 82
- pmstore function, 13, 19, 36, 46

- pmsys command, 52
- PMTRACE\_STATE\_API flag, 119
- PMTRACE\_STATE\_ASYNC flag, 119
- PMTRACE\_STATE\_COMMS flag, 119
- PMTRACE\_STATE\_NOAGENT flag, 119
- PMTRACE\_STATE\_NONE flag, 119
- PMTRACE\_STATE\_PDU flag, 119
- PMTRACE\_STATE\_PDUBUF flag, 119
- pmtraceabort function, 115
- pmtracebegin function, 115
- pmtracend function, 115
- pmtraceobs function, 117
- pmtracepoint function, 116, 117
- pmtracestate call, 119
- pmTraversePMNS function, 68, 72
- pmTrimNameSpace function, 68, 72
- pmTypeStr function, 31, 95
- pmUnitsStr function, 95
- pmUnloadNameSpace function, 69
- pmUseContext function, 73, 74
- pmUseZone function, 80
- pmview command, 1, 12, 52
- pmWhichContext function, 74
- pmWhichZone function, 80
- point tracing, 116
- program evolution, 103
- programming components, 1
- protocol data units
  - See "PDU", 12

## R

- record-mode services, 83
- removal script, 52
- restarting pmcd, 51
- retrospective analysis, 6
- ring buffers, 111
- rolling-window sampling, 109, 110

## S

- sample duration, 110, 113
- sampling techniques, 109
- scale and dimensionality, 58
- semantic types, 21
- sequential log files, 11
- service time, 120
- shell process, 8
- simple periodic sampling, 109
- simple PMDA
  - 2 branches, 4 metrics, 28
  - as daemon, 15
  - DSO, 14
  - initialization, 42
  - pmdaFetch callback, 33
- simple.color metric, 34
- simple.now metric, 35
- simple.store metric, 36
- simple.time metric, 35
- simple\_init function, 14, 33, 42
- simple\_store function, 46
- snapshot files, 11
- specific instance domain, 72
- sproc man page, 27
- state flags, 118, 119
- storage of metrics, 19
- symbolic association, 101
- synchronous protocol, 119

## T

- target domain, 11, 18, 26
- TCP/IP, 118, 123
- testing and debugging, 45
- threaded applications, 9
- time control services, 90
- timezone services, 79
- tool configuration, 52
- toolkit approach, 121



- trace facilities, 1
- trace PMDA
  - command-line options, 113
  - description, 107
  - design, 108
- trace.control.reset metric, 114
- trace.observe metrics, 117
- trace.observe.rate metric, 109
- trace.point.count metric, 116
- trace.point.rate metric, 116, 109
- trace.transact.ave\_time metric, 115, 109
- trace.transact.count metric, 115
- trace.transact.max\_time metric, 115, 109
- trace.transact.min\_time metric, 115, 109
- trace.transact.rate metric, 115, 109
- trace.transact.total\_time metric, 115
- transactions, 115
- trivial PMDA

- callbacks, 33
- initialization, 42
- singular metric, 21
- trivial\_init function, 33, 42
- two or three dimensional arrays, 22
- type field, 31

## U

- unavailable metrics support, 31

## W

- working buffers, 109, 111