# Performance Co-Pilot™ Programmer's Guide

**Performance Co-Pilot™ Programmer's Guide**
Maintained by:
The Performance Co-Pilot Development Team


<pcp@groups.io>


https://pcp.io 




## LICENSE



## TRADEMARKS AND ATTRIBUTIONS

# Table of Contents

# List of Figures

# List of Tables

# List of Examples

# About This Guide

## Table of Contents



This guide describes how to program the Performance Co-Pilot (PCP) performance analysis toolkit. PCP provides a systems-level suite of tools that cooperate to deliver distributed performance monitoring and performance management services spanning hardware platforms, operating systems, service layers, database internals, user applications and distributed architectures.

PCP is an open source, cross-platform software package - customizations, extensions, source code inspection, and tinkering in general is actively encouraged.

“About This Guide” includes short descriptions of the chapters in this book, directs you to additional sources of information, and explains typographical conventions.

# What This Guide Contains

This guide contains the following chapters:

- Chapter 1, *Programming Performance Co-Pilot*, contains a thumbnail sketch of how to program the various PCP components.
- Chapter 2, *Writing a PMDA*, describes how to write Performance Metrics Domain Agents (PMDAs) for PCP.
- Chapter 3, *PMAPI--The Performance Metrics API*, describes the interface that allows you to design custom performance monitoring tools.
- Chapter 4, *Instrumenting Applications*, introduces techniques, tools and interfaces to assist with exporting performance data from within applications.
- Appendix A, *Acronyms*, 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 wish to integrate performance data from within their applications into the PCP framework

This book is written for those who are competent with the C programming language, the UNIX or the Linux operating systems, 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* is a companion document to the *Performance Co-Pilot Programmer's Guide*, and is intended for system administrators and performance analysts who are directly using and administering PCP installations.

The *Performance Co-Pilot Tutorials and Case Studies* provides a series of real-world examples of using various PCP tools, and lessons learned from deploying the toolkit in production environments. It serves to provide reinforcement of the general concepts discussed in the other two books with additional case studies, and in some cases very detailed discussion of specifics of individual tools.

Additional resources include man pages and the project web site.

# Man Pages

The operating system man pages provide concise reference information on the use of 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, start from the following command:

```
man PCPIntro
```

Each man page usually has a "SEE ALSO" section, linking to other, related entries.

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

```
man pcp
```

The man pages are arranged in different sections separating commands, programming interfaces, and so on. For a complete list of manual sections on a platform enter the command:

```
man man
```

When referring to man pages, this guide follows a standard convention: the section number in parentheses follows the item. For example, **pminfo(1)** refers to the man page in section 1 for the **pminfo** command.

# Web Site

The following web site is accessible to everyone:

| URL | Description |
| --- | --- |
| https://pcp.io | PCP is open source software released under the GNU General Public License (GPL) and GNU Lesser General Public License (LGPL) |

# Conventions

The following conventions are used throughout this document:

| Convention | Meaning |
|---|---|
| `${PCP_VARIABLE}` | A brace-enclosed all-capital-letters syntax indicates a variable that has been sourced from the global `/etc/pcp.conf` file. These special variables indicate parameters that affect all PCP commands, and are likely to be different between platforms. |
| `command` | This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures. |
| *`variable`* | Italic typeface denotes variable entries and words or concepts being defined. |
| **`user input`** | 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 commands surround man page section numbers. |

# Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact the PCP maintainers using either the email address or the web site listed earlier.

We value your comments and will respond to them promptly.

# Chapter 1. Programming Performance Co-Pilot

## Table of Contents



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 the 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 operating system 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, *Writing a PMDA*)
- By creating new analysis or visualization tools using documented functions from the Performance Metrics Application Programming Interface (PMAPI) (Chapter 3, *PMAPI--The Performance Metrics API*)
- By adding performance instrumentation to an application using facilities from PCP libraries, which offer both sampling and event tracing models.

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

# PCP Architecture

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

PCP consists of numerous monitoring and collecting tools. `Monitoring tools` such as **pmval** and **pminfo** report on metrics, but have minimal interaction with target systems. `Collection tools`, called PMDAs, extract performance values from target systems, but do not provide 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

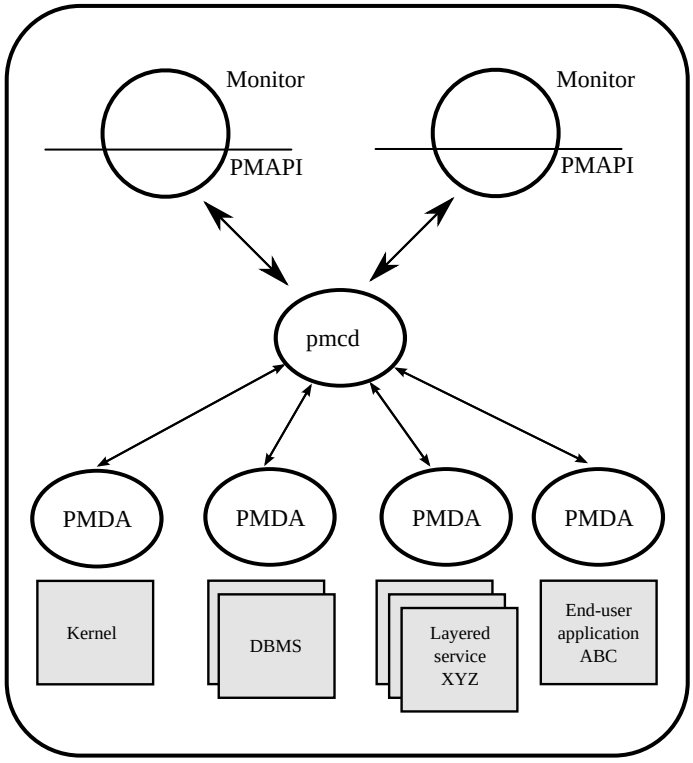
Each PCP enabled host can operate as a collector, or a monitor, or both.

Figure 1.1, “PCP Global Process Architecture” 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 the operating system kernel or a database management system) 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**

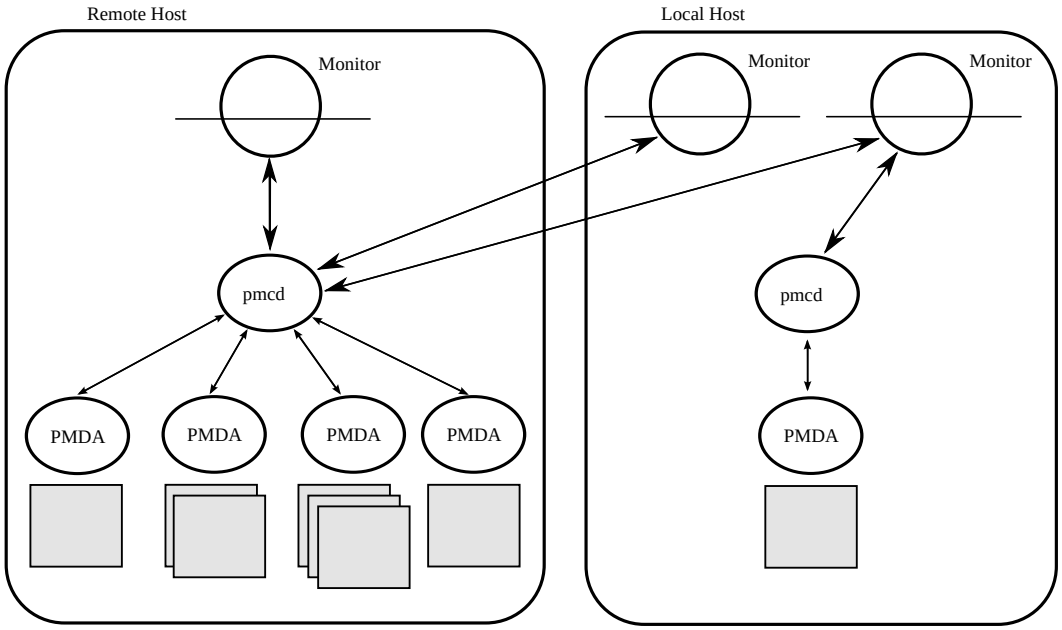



# Distributed Collection

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 operating system for which it is collecting performance measurements; there are some notable PMDAs such as Cisco and Cluster that are exceptions, and collect performance data from remote systems.

As shown in Figure 1.2, “Process Structure for Distributed Operation”, 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.

# Name Space

Each PMDA provides a domain of metrics, whether they be for the operating 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.

# Distributed PMNS

Performance metric namespace (PMNS) operations are directed by default to the host or set of archives that is the source of the desired performance metrics.

In Figure 1.2, “Process Structure for Distributed Operation”, 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.

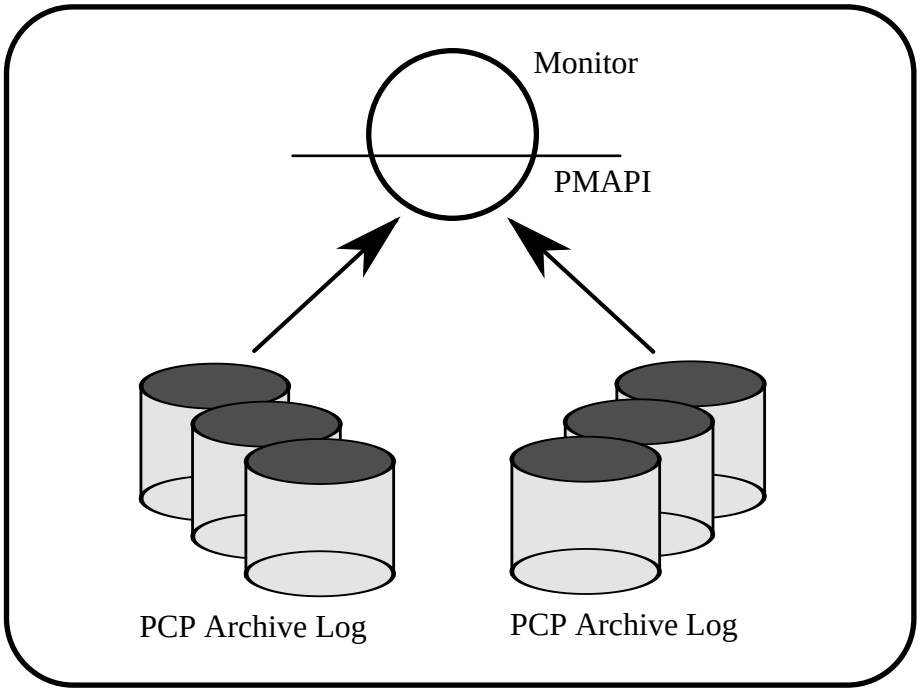
Although extremely rarely used, the **-n** *pmnsfile* command line option may be used with many PCP monitoring tools to force use of a local PMNS file in preference to the PMNS at the source of the metrics.

# 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, “Architecture for Retrospective Analysis”.

**Figure 1.3. Architecture for Retrospective Analysis**




# Overview of Component Software

Performance Co-Pilot (PCP) is composed of text-based tools, optional graphical tools, and related commands. Each tool or command is fully documented by a man page. These man pages are named after the tools or commands they describe, and are accessible through the **man** command. For example, to see the **pminfo(1)** man page for the **pminfo** command, enter this command:

```
man pminfo
```

A list of PCP developer tools and commands, grouped by functionality, is provided in the following section.

## Application and Agent Development

The following PCP tools aid the development of new programs to consume performance data, and new agents to export performance data within the PCP framework:

**chkhelp** Checks the consistency of performance metrics help database files.

**dbpmda** Allows PMDA behavior to be exercised and tested. It is an interactive debugger for PMDAs.

**mmv** Is used to instrument applications using Memory Mapped Values (MMV). These are values that are communicated with **pmcd** instantly, and very efficiently, using a shared memory mapping. It is a program instrumentation library.

**newhelp** Generates the database files for one or more source files of PCP help text.

**pmapi** Defines a procedural interface for developing PCP client applications. It is the Performance Metrics Application Programming Interface (PMAPI).

**pmclient** Is a simple client that uses the PMAPI to report some high-level system performance metrics. The source code for **pmclient** is included in the distribution.

**pmda** Is a library used by many shipped PMDAs to communicate with a **pmcd** process. It can expedite the development of new and custom PMDAs.

**pmgenmap** Generates C declarations and `cpp` macros to aid the development of customized programs that use the facilities of PCP. It is a program development tool.

# 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, *Writing a PMDA*, is a step-by-step guide to writing your own PMDA.

## 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.

# 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*.

### Note

It is strongly recommended that code for a new PMDA be based on the source of one of the existing PMDAs below the `${PCP_PMDAS_DIR}` directory.

## In-Process (DSO) Method

This method of building a PMDA uses a Dynamic Shared Object (DSO) that is attached by PMCD, using the platform-specific shared library manipulation interfaces such as **dlopen(3)**, 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 (**pmcd** does not run as root), 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 great care is required when crafting a PMDA in this manner. Calls to **exit(1)** in the PMDA, or a library it uses, would cause PMCD to exit and end monitoring of that host. Other implications are discussed in the section called “Daemon PMDA”.

## 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. For some very basic examples, see the `${PCP_PMDAS_DIR}/trivial/trivial.c` and `${PCP_PMDAS_DIR}/simple/simple.c` source files.

The daemon PMDA is actually the most common, because it allows multiple threads of control, greater (different user) privileges when executing, and provides more resilient error encapsulation than the DSO method.

### Note

Of particular interest for daemon PMDA writers, the `${PCP_PMDAS_DIR}/simple` PMDA has implementations in C, Perl and Python.

# 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, *PMAPI--The Performance Metrics API*. The PMAPI, which provides performance tool developers with access to all of the historical and live distributed services of PCP, is the interface used by the standard PCP utilities.

# Library Reentrancy and Threaded Applications

While the core PCP library (**libpcp**) is thread safe, the layered PMDA library (**libpcp_pmda**) is not. 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 PMDA library.

# Chapter 2. Writing a PMDA

## Table of Contents



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.

# 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).

### Note

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, “Creating a PMDA” 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, **pthreads** or single threaded).
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). These concepts will be further expanded in the section called “Domains, Metrics, Instances and Labels”
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. Where appropriate, define **pmie** rule templates suitable for alerting or notification systems. For more information, see the **pmie(1)** and **pmieconf(1)** man pages.
12. Where appropriate, define **pmlogger** configuration templates suitable for creating PCP archives containing the new metrics. For more information, see the **pmlogconf(1)** and **pmlogger(1)** man pages.

# 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).

# 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 a daemon PMDA, the following request PDUs must be supported:

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

The following request PDUs may optionally be supported:

| | |
|---|---|
| `PDU_PMNS_NAMES` | Request for metric names, given one or more identifiers (see the **pmLookupName(3)** man page.) |
| `PDU_PMNS_CHILD` | A list of immediate descendent nodes of a given namespace node (see the **pmGetChildren(3)** man page). |
| `PDU_PMNS_TRAVERSE` | Request for a particular sub-tree of a given namespace node (see the **pmTraversePMNS(3)** man page). |
| `PDU_PMNS_IDS` | Perform a reverse name lookup, mapping a metric identifier to a name (see the **pmNameID(3)** man page). |
| `PDU_ATTR` | Handle connection attributes (key/value pairs), such as client credentials and other authentication information (see the **__pmParseHostAttrsSpec(3)** man page). |
| `PDU_LABEL_REQ` | Request for metric labels (see the **pmLookupLabels(3)** 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.

# 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 a platform-specific call, such as **dlopen**; see the **dlopen(3)** man

page. This eliminates the need for an IPC layer (typically a 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 the section called “PMDA Interface”.

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, `${PCP_PMCDCONF_PATH}`, 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 the section called “Initializing a PMDA”.

As superuser, install the simple PMDA as a DSO, as shown in Example 2.1, “Simple PMDA as a DSO”, and observe the changes in the PMCD configuration file. The output may differ slightly depending on the operating system you are using, any other PMDAs you have installed or any PMCD access controls you have in place.

### Example 2.1. Simple PMDA as a DSO

```
              cat ${PCP_PMCDCONF_PATH}
# Performance Metrics Domain Specifications
#
# This file is automatically generated during the build
# Name  Id      IPC     IPC Params      File/Cmd
root    1       pipe    binary          /var/lib/pcp/pmdas/root/pmdaroot
pmcd    2       dso     pmcd_init       ${PCP_PMDAS_DIR}/pmcd/pmda_pmcd.so
proc    3       pipe    binary          ${PCP_PMDAS_DIR}/linux/pmda_proc.so -d 3
linux   60      dso     linux_init      ${PCP_PMDAS_DIR}/linux/pmda_linux.so
mmv     70      dso     mmv_init        /var/lib/pcp/pmdas/mmv/pmda_mmv.so
simple  254     dso     simple_init     ${PCP_PMDAS_DIR}/simple/pmda_simple.so
```

As can be seen from the contents of `${PCP_PMCDCONF_PATH}`, 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 254, as shown in the column headed `Id`.

#### Note

For some platforms the DSO file name will not be `pmda_simple.so`. On Mac OS X it is `pmda_simple.dylib` and on Windows it is `pmda_simple.dll`.

# 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, which affects all connected client tools.
- 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.
- PMCD runs as the "pcp" user; so all DSO PMDAs must also run as this user.
- A memory leak in a DSO PMDA also causes a memory leak for PMCD.

Consequently, many PMDAs 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** (or **CreateProcess** on Windows). 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 IPv4 or IPv6 TCP/IP, or UNIX domain sockets if the platform supports that; see the **tcp(7)**, **ip(7)**, **ipv6(7)** or **unix(7)** man pages.

As superuser, install the simple PMDA as a daemon process as shown in Example 2.2, “Simple PMDA as a Daemon”. Again, the output may differ due to operating system differences, other PMDAs already installed, or access control sections in the PMCD configuration file.

### 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 `${PCP_PMDAS_DIR}/simple/pmdasimple`, using domain number 253.

```
# cd ${PCP_PMDAS_DIR}/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 ${PCP_PMCDCONF_PATH}
# Performance Metrics Domain Specifications
#
# This file is automatically generated during the build
# Name  Id      IPC     IPC Params      File/Cmd
root    1       pipe    binary          /var/lib/pcp/pmdas/root/pmdaroot
pmcd    2       dso     pmcd_init       ${PCP_PMDAS_DIR}/pmcd/pmda_pmcd.so
proc    3       pipe    binary          ${PCP_PMDAS_DIR}/linux/pmda_proc.so -d 3
linux   60      dso     linux_init      ${PCP_PMDAS_DIR}/linux/pmda_linux.so
mmv     70      dso     mmv_init        /var/lib/pcp/pmdas/mmv/pmda_mmv.so
simple  253     pipe    binary          ${PCP_PMDAS_DIR}/simple/pmdasimple -d 253
```

# 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 `${PCP_PMDAS_DIR}/cisco` directory and the **pmdacisco(1)** man page.

# Domains, Metrics, Instances and Labels

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. Refer to the `${PCP_PMDAS_DIR}/trivial` and `${PCP_PMDAS_DIR}/simple` directories, respectively, where both binaries and source code are available.

# 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.

# 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 `${PCP_VAR_DIR}/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 `${PCP_PMCDCONF_PATH}` control file used by PMCD is all that is required for successful starting of PMCD and the associated PMDAs, the developer of a new PMDA is encouraged to add the default domain number for each new PMDA to the `${PCP_VAR_DIR}/pmns/stdpmid.local` file and then to run the `Make.stdpmid` script in `${PCP_VAR_DIR}/pmns` to recreate `${PCP_VAR_DIR}/pmns/stdpmid`; this file acts as a repository for documenting the known default domain numbers.

# 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 (`hinv.ninterface`, `network.interface.speed`, `network.interface.duplex`, and so on)

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.

## 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:

`pmid` A unique identifier, Performance Metric Identifier (PMID), that differentiates this metric from other metrics across the union of all PMDAs

`type` 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

`indom` An instance domain identifier that links this metric to an instance domain

`sem` An encoding of the value's semantics (counter, instantaneous, or discrete)

`units` A description of the value's units based on dimension and scale in the three orthogonal dimensions of space, time, and count (or events)

## Note

This information can be observed for metrics from any active PMDA using **pminfo** command line options, for example:

```
$ pminfo -d -m network.interface.out.drops

network.interface.out.drops PMID: 60.3.11
    Data Type: 64-bit unsigned int  InDom: 60.3 0xf000003
    Semantics: counter  Units: count
```

Symbolic constants of the form `PM_TYPE_*`, `PM_SEM_*`, `PM_SPACE_*`, `PM_TIME_*`, and `PM_COUNT_*` are defined in the `<pcp/pmapi.h>` header 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, “ `__pmID_int` Structure”:

### Example 2.3. `__pmID_int` Structure

```
typedef struct {
        int             flag:1;
        unsigned int    domain:9;
        unsigned int    cluster:12;
        unsigned int    item:10;
} __pmID_int;
```

For additional information, see the `<pcp/libpcp.h>` file.

The `flag` field should be ignored. The `domain` number should be set at run time when the PMDA is initialized. The `PMDA_PMID` macro defined in `<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 the section called “Name Space”.

A table of `pmdaMetric` structures should be defined within the PMDA, with one structure per metric as shown in Example 2.4, “ `pmdaMetric` Structure”.

### 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, “Trivial PMDA”, has only a singular metric (that is, no instance domain):

### Example 2.5. Trivial PMDA

```
static pmdaMetric metrictab[] = {
/* time */
 { NULL,
   { 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. Note that this is not yet a complete PMID, the domain number which identifies the PMDA will be combined with it at runtime.
- 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”

## 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 non-numeric values).

For a given time interval covering six consecutive timestamps, each spanning two units of time, the metric values in Example 2.6, “Effect of Semantics on a Metric” 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
```

Note that these interpretations of metric semantics are performed by the monitor tool, automatically, before displaying a value and they are not transformations that the PMDA performs.

# 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, as described in the section called “Instance Identification”.

## Instance Identification

Consistent interpretation of instances and instance domains require a few simple rules to be followed by PMDA authors. The PMDA library provides a series of **pmdaCache** routines to assist.

- Each internal instance identifier (numeric) must be a unique 31-bit number.
- The external instance name (string) must be unique.
- When the instance name contains a space, the name to the left of the first space (the short name) must also be unique.
- Where an external instance name corresponds to some object or entity, there is an expectation that the association between the name and the object is fixed.
- It is preferable, although not mandatory, for the association between and external instance name (string) and internal instance identifier (numeric) to be persistent.

## 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 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. The use of metric label metadata - arbitrary key/value pairs - allows the implementer to capture the higher dimensions of the performance data.

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_TYPE_AGGREGATE`). 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. The usual issues of platform-depdendence must also be kept in mind for this case - endianness, word-size, alignment and so on - the (possibly remote) special PMAPI client tools may need this information in order to decode the data successfully.

## 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, “ `pmdaInstid` Structure”:

### 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, “ `pmdaIndom` Structure”:

### 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, “ `__pmInDom_int` Structure” 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             flag:1;
```

```
        unsigned int    domain:9;   /* 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, “Simple PMDA”, 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_PMID(0, 0), PM_TYPE_U32, PM_INDOM_NULL, PM_SEM_INSTANT,
       PMDA_PMUNITS(0, 0, 0, 0, 0, 0) }, },
/* color */
    { NULL,
      { PMDA_PMID(0, 1), PM_TYPE_32, COLOR_INDOM, PM_SEM_INSTANT,
        PMDA_PMUNITS(0, 0, 0, 0, 0, 0) }, },
/* time.user */
    { NULL,
      { PMDA_PMID(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_PMID(1,3), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        PMDA_PMUNITS(0, 1, 0, 0, PM_TIME_SEC, 0) }, },
/* now */
    { NULL,
```

```
      { PMDA_PMID(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 `metrictab[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.any.fetch = simple_fetch;
dp->version.any.instance = simple_instance;
```

# Labels

Metrics and instances can be further described through the use of metadata labels, which are arbitrary name:value pairs associated with individual metrics and instances. There are several applications of this concept, but one of the most important is the ability to differentiate the components of a multi-dimensional instance name, such as the case of the `mem.zoneinfo.numa_hit` metric which has one value per memory zone, per NUMA node.

Consider Example 2.11, “Multi-dimensional Instance Domain Labels”:

### Example 2.11. Multi-dimensional Instance Domain Labels

```
$ pminfo -l mem.zoneinfo.numa_hit

mem.zoneinfo.numa_hit
   inst [0 or "DMA::node0"] labels {"device_type":["numa_node","memory"],"indom_n
   inst [1 or "Normal::node0"] labels {"device_type":["numa_node","memory"],"indo
   inst [2 or "DMA::node1"] labels {"device_type":["numa_node","memory"],"indom_n
   inst [3 or "Normal::node1"] labels {"device_type":["numa_node","memory"],"indo
```

#### Note

The metric labels used here individually describe the memory zone and NUMA node associated with each instance.

The PMDA implementation is only partially responsible for choosing the label identifiers that differentiate components of metrics and instances within an instance domain. Label sets for a singleton metric or

individual instance of a set-valued metric are formed from a label hierarchy, which includes global labels applied to all metrics and instances from one PMAPI context.

Labels are stored and communicated within PCP using JSONB format. This format is a restricted form of JSON suitable for indexing and other operations. In JSONB form, insignificant whitespace is discarded, and the order of label names is not preserved. Within the PMCS a lexicographically sorted key space is always maintained, however. Duplicate label names are not permitted. The label with highest precedence is the only one presented. If duplicate names are presented at the same hierarchy level, only one will be preserved (exactly which one wins is arbitrary, so do not rely on this).

## Label Hierarchy

The set of labels associated with any singleton metric or instance is formed by merging the sets of labels at each level of a hierarchy. The lower levels of the hierarchy have highest precendence when merging overlapping (duplicate) label names:

- Global context labels (as reported by the `pmcd.labels` metric) are the lowest precedence. The PMDA implementor has no influence over labels at this level of the hierarchy, and these labels are typically supplied by **pmcd** from `/etc/pcp/labels` files.
- Domain labels, for all metrics and instances of a PMDA, are the next highest precedence.
- Instance Domain labels, associated with an InDom, are the next highest precedence.
- Metric cluster labels, associated with a PMID cluster, are the next highest precedence.
- Metric item labels, associated with an individual PMID, are the next highest precedence.
- Instance labels, associated with a metric instance identifier, have the highest precedence.

## Data Structures

In any PMDA that supports labels at any level of the hierarchy, each individual label (one name:value pair) requires a `pmLabel` structure as shown in Example 2.12, “ `pmLabel` Structure”:

### Example 2.12. `pmLabel` Structure

```
typedef struct {
    uint     name : 16;      /* label name offset in JSONB string */
    uint     namelen : 8;    /* length of name excluding the null */
    uint     flags : 8;      /* information about this label */
    uint     value : 16;     /* offset of the label value */
    uint     valuelen : 16;  /* length of value in bytes */
} pmLabel;
```

The `flags` field is a bitfield identifying the hierarchy level and whether this name:value pair is intrinsic (optional) or extrinsic (part of the mandatory, identifying metadata for the metric or instance). All other fields are offsets and lengths in the JSONB string from an associated `pmLabelSet` structure.

Zero or more labels are specified via a label set, in a `pmLabelSet` structure as shown in Example 2.13, “ `pmLabelSet` Structure”:

### Example 2.13. `pmLabelSet` Structure

```
typedef struct {
```

```
    uint     inst;          /* PM_IN_NULL or the instance ID */
    int      nlabels;       /* count of labels or error code */
    char     *json;         /* JSONB formatted labels string */
    uint     jsonlen : 16;  /* JSON string length byte count */
    uint     padding : 16;  /* zero, reserved for future use */
    pmLabel  *labels;       /* indexing into the JSON string */
} pmLabelSet;
```

This provides information about the set of labels associated with an entity (context, domain, indom, metric cluster, item or instance). The entity will be from any one level of the label hierarchy. If at the lowest hierarchy level (which happens to be highest precedence - instances) then the `inst` field will contain an actual instance identifier instead of PM_IN_NULL.

For information about how a label can be associated with each level of the hierarchy, see the **pmdaLabel(3)** man page.

The simple PMDA, shown in Example 2.14, “Simple PMDA”, associates labels at the domain, indom and instance levels of the hierarhy.

### Example 2.14. Simple PMDA

```
static int
simple_label(int ident, int type, pmLabelSet **lpp, pmdaExt *pmda)
{
    int         serial;

    switch (type) {
    case PM_LABEL_DOMAIN:
        pmdaAddLabels(lpp, "{"role":"testing"}");
        break;
    case PM_LABEL_INDOM:
        serial = pmInDom_serial((pmInDom)ident);
        if (serial == COLOR_INDOM) {
            pmdaAddLabels(lpp, "{"indom_name":"color"}");
            pmdaAddLabels(lpp, "{"model":"RGB"}");
        }
        if (serial == NOW_INDOM) {
            pmdaAddLabels(lpp, "{"indom_name":"time"}");
            pmdaAddLabels(lpp, "{"unitsystem":"SI"}");
        }
        break;
    case PM_LABEL_CLUSTER:
    case PM_LABEL_ITEM:
        /* no labels to add for these types, fall through */
    default:
        break;
    }
    return pmdaLabel(ident, type, lpp, pmda);
}

static int
simple_labelCallBack(pmInDom indom, unsigned int inst, pmLabelSet **lp)
{
    struct timeslice *tsp;
```

```
    if (pmInDom_serial(indom) != NOW_INDOM)
        return 0;
    if (pmdaCacheLookup(indom, inst, NULL, (void *)&tsp) != PMDA_CACHE_ACTIVE)
        return 0;
    /* SI units label, value: sec (seconds), min (minutes), hour (hours) */
    return pmdaAddLabels(lp, "{"units":"%s"}", tsp-<tm_name);
}
```

The **simple_labelCallBack** function is called indirectly via **pmdaLabel** for each instance of the NOW_INDOM. PMDA initialization ensures these functions are registered with the global PMDA interface structure for use when handling label requests, by the following assignments in the **simple_init** function:

```
dp->version.seven.label = simple_label;
pmdaSetLabelCallBack(dp, simple_labelCallBack);
```

# 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.

## 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 operating system kernel, provide a kernel interface (preferred) to export the performance data.

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

## 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 child processes to obtain information while the main process communicates with PMCD.

At the simplest deployment of this arrangement, the two processes may execute without synchronization. Threads have also been used as a more portable multithreading mechanism; see the **pthreads(7)** 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.

### Warning

Since certain data structures used by the PMDA library are not thread-aware, only one PMDA thread of control should call PMDA library functions - this would typically be the thread servicing requests from PMCD.

One caveat about this style of caching PMDA--in this (special) case it is 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). The finer points of metric semantics are discussed in the section called “Semantics”

# 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(5)** man page and its hierarchical nature, syntax, and helper tools are further described in the *Performance Co-Pilot User's and Administrator's Guide*.

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 `${PCP_VAR_DIR}/pmns/root`.

Example 2.15, “ `pmns` File for the Simple PMDA” 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 2.15. `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 2.16, “Alternate `pmns` File for the Simple PMDA” shows an alternative PMNS for the simple PMDA:

### Example 2.16. 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 ${PCP_VAR_DIR}/pmns/stdpmid for the corresponding PMDA during installation (for the simple PMDA, this would normally be the value 253).

If the PMDA implementer so chooses, all or a subset of the metric names and identifiers can be specified programatically. In this situation, a special asterisk syntax is used to denote those subtrees which are to be handles this way. Example 2.17, "Dynamic metrics pmns File for the Simple PMDA" shows this dynamic namespace syntax, for all metrics in the simple PMDA:

### Example 2.17. Dynamic metrics pmns File for the Simple PMDA

```
simple          SIMPLE:*:*
```

In this example, like the one before, the SIMPLE macro is replaced by the domain number, and all (simple.*) metric namespace operations must be handled by the PMDA. This is in contrast to the static metric name model earlier, where the host-wide PMNS file is updated and used by PMCD, acting on behalf of the agent.

## 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 2.18, "Help Text for the Simple PMDA", 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 ${PCP_PMDAS_DIR} as part of the PMDA installation procedure.

### Example 2.18. 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.0 Instance domain "colour" for simple PMDA
Universally 3 instances, "red" (0), "green" (1) and "blue" (3).

@ SIMPLE.1 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 my 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
${PCP_PMDAS_DIR}/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.
```

# 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 2.19, " Setting Values" is typically benign:

  **Example 2.19. Setting Values**

  ```
  pmDesc dummy = {
       .pmid  = PMDA_PMID(3,0),           /* pmid, fill this in */
       .type  = PM_TYPE_NOSUPPORT,        /* this is the important part */
       .indom = PM_INDOM_NULL,            /* singular,causes no problems */
       .sem   = 0,                        /* no semantics */
       .units = PMDA_PMUNITS(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_PMID`.

- 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 `PM_ERR_PERMISSION` 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.

# PMDA 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 is also used by daemon PMDAs with **pmdaMain**.

## 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 `<pcp/pmda.h>`:

- **pmdaFetch**

- **pmdaProfile**
- **pmdaInstance**
- **pmdaDesc**
- **pmdaText**
- **pmdaStore**
- **pmdaPMID**
- **pmdaName**
- **pmdaChildren**
- **pmdaAttribute**
- **pmdaLabel**

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`.

## Trivial PMDA

The trivial PMDA uses all of the default callbacks as shown in Example 2.20, “Request Handling Callbacks in the Trivial PMDA”. The additional callback for **pmdaFetch** is defined as **trivial_fetchCallBack**:

### Example 2.20. 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_PMID;
   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**. As a rule of thumb, the API routines with named ending with **CallBack** are helpers for the higher PDU handling routines like **pmdaFetch**. The latter are set directly using the PMDA Interface Structures, as described in the section called “PMDA Structures”.

## Simple PMDA

The simple PMDA callback for **pmdaFetch** is more complicated because it supports more metrics, some metrics are instantiated with each fetch, and one instance domain is dynamic. The default **pmdaFetch** callback, shown in Example 2.21, "Request Handling Callbacks in the Simple PMDA", 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 2.21. 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 available types and associated fields are described further in the section called "Performance Metric Descriptions" and Example 3.18, " `pmAtomValue` Structure".

#### Note

Note that PMID validity checking need only check the cluster and item numbers, the domain number is guaranteed to be valid and the PMDA should make no assumptions about the actual domain number being used at this point.

The `simple.numfetch` metric has no instance domain and is easily handled first as shown in Example 2.22, " `simple.numfetch` Metric":

### Example 2.22. `simple.numfetch` Metric

```
static int
simple_fetchCallBack(pmdaMetric *mdesc, unsigned int inst, pmAtomValue *atom)
{
   int             i;
   static int      oldfetch;
   static double   usr, sys;
   __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) {                   /* simple.numfetch */
           atom->l = numfetch;
       }
```

In Example 2.23, " `simple.color` Metric", the `inst` parameter is used to specify which instance is required for the `simple.color` metric:

### Example 2.23. `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_PMID;
```

In Example 2.24, " `simple.time` Metric", 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 `usr` and `sys` values should be updated:

### Example 2.24. `simple.time` Metric

```
  else if (idp->cluster == 1) {             /* simple.time */
      if (oldfetch < numfetch) {
          __pmProcessRunTimes(&usr, &sys);
          oldfetch = numfetch;
      }
      if (idp->item == 2)                   /* simple.time.user */
          atom->d = usr;
      else if (idp->item == 3)              /* simple.time.sys */
          atom->d = sys;
      else
          return PM_ERR_PMID;
   }
```

In Example 2.25, " `simple.now` Metric", the `simple.now` metric is in a third cluster and uses `inst` again to select a specific instance from the `INDOM_NOW` instance domain. The values associated with instances in this instance domain are managed using the **pmdaCache(3)** helper routines, which provide efficient interfaces for managing more complex instance domains:

### Example 2.25. `simple.now` Metric

```
   else if (idp->cluster == 2) {
       if (idp->item == 4) {                /* simple.now */
           struct timeslice *tsp;
```

```
                sts = pmdaCacheLookup(*now_indom, inst, NULL, (void *)&tsp);
                if (sts != PMDA_CACHE_ACTIVE) {
                    if (sts < 0)
                        pmNotifyErr(LOG_ERR, "pmdaCacheLookup failed: inst=%d: %s",
                                    inst, pmErrStr(sts));
                    return PM_ERR_INST;
                }
                atom->l = tsp->tm_field;
            }
            else
                return PM_ERR_PMID;
        }
```

## `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 2.26, " `simple_store` in the Simple PMDA". For additional information, see the **pmstore(1)** and **pmStore(3)** man pages.

### Example 2.26. `simple_store` in the Simple PMDA

The **pmdaStore** callback (which returns `PM_ERR_PERMISSION` 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;

    /* a store request may affect multiple metrics at once */
    for (i = 0; i < result->numpmid; i++) {
        vsp = result->vset[i];
        pmidp = (__pmID_int *)&vsp->pmid;
        if (pmidp->cluster == 0) {  /* storable metrics are cluster 0 */
            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 */
                /* a store request may affect multiple instances at once */
                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 2.27, " `simple.color` and `PM_ERR_INST` Errors":

### Example 2.27. `simple.color` and `PM_ERR_INST` Errors

```
                    switch (vsp->vlist[j].inst) {
                    case 0:                         /* red */
                        red = val;
                        break;
                    case 1:                         /* green */
                        green = val;
                        break;
                    case 2:                         /* blue */
                        blue = val;
                        break;
                    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 2.28, " `PM_ERR_PMID` Errors":

### Example 2.28. `PM_ERR_PMID` Errors

```
            default:
                sts = PM_ERR_PMID;
                break;
        }
    }
```

Any metrics that cannot be altered should generate an error value of `PM_ERR_PERMISSION`, and metrics not supported by the PMDA should result in an error value of `PM_ERR_PMID` as shown in Example 2.29, " `PM_ERR_PERMISSION` and `PM_ERR_PMID` Errors":

### Example 2.29. `PM_ERR_PERMISSION` and `PM_ERR_PMID` Errors

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

```
            sts = PM_ERR_PMID;
            break;
        }
    }
    return sts;
}
```

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

### Note

When using storable metrics, it is important to consider the implications. It is possible **pmlogger** is actively sampling the metric being modified, for example, which may cause unexpected results to be persisted in an archive. Consider also the use of client credentials, available via the **attribute** callback of the `pmdaInterface` structure, to appropriately limit access to any modifications that might be made via your storable metrics.

## 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, `PM_ERR_PMID`, `PM_ERR_INST` or `PM_ERR_AGAIN`) |
| 0 | Success |

In `PMDA_INTERFACE_3` and all later versions, the return codes for the **pmdaFetch** callback function are defined:

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

# PMDA Structures

PMDA structures used with the `pcp_pmda` library are defined in `<pcp/pmda.h>`. Example 2.30, " `pmdaInterface` Structure Header" and Example 2.32, " `pmdaExt` Stucture" describe the `pmdaInterface` and `pmdaExt` structures.

### Example 2.30. `pmdaInterface` Structure Header

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;          /* optional feature flags */
} comm;             /* set/return communication and version info */
int status;         /* return initialization status here */
union {
    ...
```

This structure is passed by PMCD to a DSO PMDA as an argument to the initialization function. This structure supports multiple (binary-compatible) versions--the second and subsequent versions have support for the `pmdaExt` structure. Protocol version one is for backwards compatibility only, and should not be used in any new PMDA.

To date there have been six revisions of the interface structure:

- Version two added the `pmdaExt` structure, as mentioned above.
- Version three changed the fetch callback return code semantics, as mentioned in the section called "Return Codes for **pmdaFetch** Callbacks".
- Version four added support for dynamic metric names, where the PMDA is able to create and remove metric names on-the-fly in response to changes in the performance domain (**pmdaPMID**, **pmdaName**, **pmdaChildren** interfaces)
- Version five added support for per-client contexts, where the PMDA is able to track arrival and disconnection of PMAPI client tools via PMCD (**pmdaGetContext** helper routine). At the same time, support for `PM_TYPE_EVENT` metrics was implemented, which relies on the per-client context concepts (**pmdaEvent*** helper routines).
- Version six added support for authenticated client contexts, where the PMDA is informed of user credentials and other PMCD attributes of the connection between individual PMAPI clients and PMCD (**pmdaAttribute** interface)
- Version seven added support for metadata labels, where the PMDA is able to associate name:value pairs in a hierarchy such that additional metadata, above and beyond the metric descriptors, is associated with metrics and instances (**pmdaLabel** interface)

### Example 2.31. `pmdaInterface` Structure, Latest Version

```
...
union {
    ...
    /*
     * PMDA_INTERFACE7
     */
    struct {
        pmdaExt *ext;
        int     (*profile)(pmdaInProfile *, pmdaExt *);
        int     (*fetch)(int, pmID *, pmResult **, pmdaExt *);
        int     (*desc)(pmID, pmDesc *, pmdaExt *);
        int     (*instance)(pmInDom, int, char *, pmdaInResult **, pmdaExt *);
        int     (*text)(int, int, char **, pmdaExt *);
        int     (*store)(pmResult *, pmdaExt *);
        int     (*pmid)(const char *, pmID *, pmdaExt *);
        int     (*name)(pmID, char ***, pmdaExt *);
        int     (*children)(const char *, int, char ***, int **, pmdaExt *);
        int     (*attribute)(int, int, const char *, int, pmdaExt *);
```

```
            int     (*label)(int, int, pmLabelSet **, pmdaExt *);
        } seven;
    } version;
} pmdaInterface;
```

## Note

Each new interface version is always defined as a superset of those that preceded it, only adds fields at the end of the new structure in the union, and is always binary backwards-compatible. `And thus it shall remain.` For brevity, we have shown only the latest interface version (seven) above, but all prior versions still exist, build, and function. In other words, PMDAs built against earlier versions of this header structure (and PMDA library) function correctly with the latest version of the PMDA library.

### Example 2.32. `pmdaExt` Stucture

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

```
typedef struct {
    unsigned int e_flags;       /* PMDA_EXT_FLAG_* bit field */
    void        *e_ext;         /* used internally within libpcp_pmda */
    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 */
    /* added for PMDA_INTERFACE_5 */
    int         e_context;      /* client context id from pmcd */
    pmdaEndContextCallBack e_endCallBack;  /* callback after client context closed
    /* added for PMDA_INTERFACE_7 */
    pmdaLabelCallBack  e_labelCallBack;  /* callback to lookup metric instance lab
} 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)**, **pmdaGetOptions(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.

# 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.

## 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.

## Common Initialization

As described in the section called “DSO PMDA”, 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.

### Trivial PMDA

Example 2.33, “Initialization in the Trivial PMDA” 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 2.33. Initialization in the Trivial PMDA**

```
static char     *username;
static int      isDSO = 1;              /* ==0 if I am a daemon */

void trivial_init(pmdaInterface *dp)
{
```

```
    if (isDSO)
        pmdaDSO(dp, PMDA_INTERFACE_2, “trivial DSO”,
                “${PCP_PMDAS_DIR}/trivial/help”);
    else
        pmSetProcessIdentity(username);

    if (dp->status != 0)
        return;

    pmdaSetFetchCallBack(dp, trivial_fetchCallBack);
    pmdaInit(dp, NULL, 0,
             metrictab, sizeof(metrictab)/sizeof(metrictab[0]));
}
```

The trivial PMDA can execute as either a DSO or daemon PMDA. A default installation installs it as a daemon, however, and the **main** routine clears *isDSO* and sets *username* accordingly.

The **trivial_init** routine provides the opportunity to do any extra DSO or daemon setup before calling the library **pmdaInit**. In the example, the help text is setup for DSO mode and the daemon is switched to run as an unprivileged user (default is `root`, but it is generally good form for PMDAs to run with the least privileges possible). If `dp->status` is non-zero after the **pmdaDSO** call, the PMDA will be removed by PMCD and cannot safely continue to use the **pmdaInterface** structure.

## Simple PMDA

In Example 2.34, “Initialization in the Simple PMDA”, 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 2.34. Initialization in the Simple PMDA

```
static int      isDSO = 1;              /* =0 I am a daemon */
static char     *username;

void simple_init(pmdaInterface *dp)
{
    if (isDSO)
        pmdaDSO(dp, PMDA_INTERFACE_7, “simple DSO”,
                “${PCP_PMDAS_DIR}/simple/help”);
    else
        pmSetProcessIdentity(username);

    if (dp->status != 0)
        return;

    dp->version.any.fetch = simple_fetch;
    dp->version.any.store = simple_store;
    dp->version.any.instance = simple_instance;
    dp->version.seven.label = simple_label;
    pmdaSetFetchCallBack(dp, simple_fetchCallBack);
    pmdaSetLabelCallBack(dp, simple_labelCallBack);
    pmdaInit(dp, indomtab, sizeof(indomtab)/sizeof(indomtab[0]),
             metrictab, sizeof(metrictab)/sizeof(metrictab[0]));
}
```

Once again, 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 always changes the value of this variable to 0 in `main`, for PMDAs that can operate in both modes.

Remember also, as described earlier, **simple_fetch** is dealing with a single request for (possibly many) values for metrics from the PMDA, and **simple_fetchCallBack** is its little helper, dealing with just one metric and one instance (optionally, if the metric happens to have an instance domain) within that larger request.

# 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)**, **pmdaGetOptions(3)**, **pmdaOpenLog(3)**, **pmdaConnect(3)**, and **pmdaMain(3)** man pages.

## Note

Optionally, a daemon PMDA may wish to reduce or change its privilege level, as seen in Example 2.33, “Initialization in the Trivial PMDA” and Example 2.34, “Initialization in the Simple PMDA”. Some performance domains `require` the extraction process to run as a specific user in order to access the instrumentation. Many domains require the default `root` level of access for a daemon PMDA.

The simple PMDA specifies the command-line arguments it accepts using **pmdaGetOptions**, as shown in Example 2.35, “ `main` in the Simple PMDA”. For additional information, see the **pmdaGetOptions(3)** man page.

### Example 2.35. `main` in the Simple PMDA

```
static pmLongOptions longopts[] = {
    PMDA_OPTIONS_HEADER(“Options”),
    PMOPT_DEBUG,
    PMDAOPT_DOMAIN,
    PMDAOPT_LOGFILE,
    PMDAOPT_USERNAME,
    PMOPT_HELP,
    PMDA_OPTIONS_TEXT(“\nExactly one of the following options may appear:”),
    PMDAOPT_INET,
    PMDAOPT_PIPE,
    PMDAOPT_UNIX,
    PMDAOPT_IPV6,
    PMDA_OPTIONS_END
};
```

```
static pmdaOptions opts = {
    .short_options = “D:d:i:l:pu:U:6:?”,
    .long_options = longopts,
};

int
main(int argc, char **argv)
{
    pmdaInterface       dispatch;

    isDSO = 0;
    pmSetProgname(argv[0]);
    pmGetUsername(&username);
    pmdaDaemon(&dispatch, PMDA_INTERFACE_7, pmGetProgname(), SIMPLE,
               “simple.log”, “${PCP_PMDAS_DIR}/simple/help”);

    pmdaGetOptions(argc, argv, &opts, &dispatch);
    if (opts.errors) {
        pmdaUsageMessage(&opts);
        exit(1);
    }
    if (opts.username)
        username = opts.username;

    pmdaOpenLog(&dispatch);
    simple_init(&dispatch);
    simple_timenow_check();
    pmdaConnect(&dispatch);
    pmdaMain(&dispatch);

    exit(0);
}
```

The conditions under which **pmdaMain** will return are either unexpected error conditions (often from failed initialisation, which would already have been logged), or when PMCD closes the connection to the PMDA. In all cases the correct action to take is simply to exit cleanly, possibly after any final cleanup the PMDA may need to perform.

# 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.

## 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.

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

${PCP_PMCDOPTIONS_PATH}, attaching a debugger (such as **gdb**) to the PMDA process 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 ${PCP_LOG_DIR}/pmcd/pmcd.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.

# Debugging Information

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

The PMCD debugging options 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 libpmda option is set within the PMDA; see the **PMDA(3)** man page. The command-line argument **-D** is trapped by **pmdaGetOptions** to set the global debugging control options. Adding tests within the PMDA for the appl0, appl1 and 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.

Adding this segment of code to the **simple_store** metric causes a timestamped log message to be sent to the current log file whenever **pmstore** attempts to change simple.numfetch and the PCP debugging options have the appl0 option set as shown in Example 2.36, “ simple.numfetch in the Simple PMDA”:

### Example 2.36. simple.numfetch in the Simple PMDA

```
case 0: /* simple.numfetch */
     x
     val = vsp->vlist[0].value.lval;
     if (val < 0) {
         sts = PM_ERR_SIGN;
         val = 0;
     }
     if (pmDebugOptions.appl0__) {
         pmNotifyErr(LOG_DEBUG,
               "simple: %d stored into numfetch", val);
     }
     numfetch = val;
     break;
```

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

## 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 a debugger, tracing utilities, and other diagnostic tools. The **dbpmda(1)** man page contains a sample session with the `simple` PMDA.

# Integration of a 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 `${PCP_PMDAS_DIR}` into which several files would be installed. In the description in the section called "Installing a PMDA", the PMDA of interest is assumed to be known by the name `newbie`, hence the PMDA directory would be `${PCP_PMDAS_DIR}/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.

## 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 `${PCP_PMDAS_DIR}/newbie`.
- The name space has been updated in the `${PCP_VAR_DIR}/pmns` directory.
- The PMDA binary has been installed, usually in the directory `${PCP_PMDAS_DIR}/newbie`.
- The `${PCP_PMCDCONF_PATH}` file has been updated.
- The PMCD process has been restarted or notified (with a `SIGHUP` signal) that the new PMDA exists.

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. The `clobber` target should remove any files created as a by-product of the `install` target.

You may wish to use `${PCP_PMDAS_DIR}/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.

The `Install` script should make use of the generic procedures defined in the script `${PCP_SHARE_DIR}/lib/pmdaproc.sh`, and may be as straightforward as the one used for the trivial PMDA, shown in Example 2.37, " `Install` Script for the Trivial PMDA":

### Example 2.37. `Install` Script for the Trivial PMDA

```
. ${PCP_DIR}/etc/pcp.env
```

```
. ${PCP_SHARE_DIR}/lib/pmdaproc.sh

iam=trivial
pmdaSetup
pmdainstall
exit
```

The variables, shown in Table 2.1, “Variables to Control Behavior of Generic `pmdaproc.sh` Procedures”, may be assigned values to modify the behavior of the `pmdaSetup` and `pmdainstall` procedures from `${PCP_SHARE_DIR}/lib/pmdaproc.sh`.

**Table 2.1. Variables to Control Behavior of Generic `pmdaproc.sh` Procedures**

| Shell Variable | Use | Default |
|---|---|---|
| `$iam` | Name of the PMDA; assignment to this variable is mandatory.<br><br>Example: `iam=newbie` | |
| `$dso_opt` | Can this PMDA be installed as a DSO? | `false` |
| `$daemon_opt` | Can this PMDA be installed as a daemon? | `true` |
| `$perl_opt` | Is this PMDA a perl script? | `false` |
| `$python_opt` | Is this PMDA a python script? | `false` |
| `$pipe_opt` | If installed as a daemon PMDA, is the default IPC via pipes? | `true` |
| `$socket_opt` | If installed as a daemon PMDA, is the default IPC via an Internet socket? | `false` |
| `$socket_inet_def` | If installed as a daemon PMDA, and the IPC method uses an Internet socket, the default port number. | |
| `$ipc_prot` | IPC style for PDU exchanges involving a daemon PMDA; `binary` or `text`. | `binary` |
| `$check_delay` | Delay in seconds between installing PMDA and checking if metrics are available. | `3` |
| `$args` | Additional command-line arguments passed to a daemon PMDA. | |
| `$pmns_source` | The name of the PMNS file (by default relative to the PMDA directory). | `pmns` |
| `$pmns_name` | First-level name for this PMDA's metrics in the PMNS. | `$iam` |
| `$help_source` | The name of the help file (by default relative to the PMDA directory). | `help` |
| `$pmda_name` | The name of the executable for a daemon PMDA. | `pmda$iam` |
| `$dso_name` | The name of the shared library for a DSO PMDA. | `pmda$iam.`<br>`$dso_suffix` |
| `$dso_entry` | The name of the initialization function for a DSO PMDA. | `${iam}_init` |
| `$domain` | The numerical PMDA domain number (from `domain.h`). | |
| `$SYMDOM` | The symbolic name of the PMDA domain number (from `domain.h`). | |

| Shell Variable | Use | Default |
|---|---|---|
| `$status` | Exit status for the shell script | `0` |

In addition, the variable `do_check` will be set to reflect the intention to check the availability of the metrics once the PMDA is installed. By default this variable is `true` however the command-line option `-Q` to `Install` may be used to set the variable to `false`.

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

The detailed operation of the `pmdainstall` 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 (`${PCP_VAR_DIR}/config/pmchart`) with the ".pmchart" suffix removed.
- Create new help files from `$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 (`${PCP_PMCDCONF_PATH}`).
- 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. However, if the newly installed PMDA must run using a different privilege level to PMCD then PMCD must be restarted. This is because PMCD runs unprivileged after initially starting the PMDAs.
- 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 the section called "Removing a PMDA", or more drastically, restart PMCD as follows:

```
# ${PCP_RC_DIR}/pcp start
```

The files `${PCP_PMDAS_DIR}/*/Install` provide a wealth of examples that may be used to construct a new PMDA `Install` script.

# Removing a PMDA

The simplest way to stop a PMDA from running, apart from killing the process, is to remove the entry from `${PCP_PMCDCONF_PATH}` 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 the section called “Installing a PMDA”.

The `${PCP_PMDAS_DIR}/*/Remove` files provide a wealth of examples that may be used to construct a new PMDA `Remove` script.

# 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.

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.

Metrics suitable for on-going logging can be specified in templated **pmlogger** configuration files for **pmlogconf** to automatically add to the **pmlogger_daily** recorded set; see the **pmlogger(1)**, **pmlogconf(1)** and **pmlogger_daily(1)** man pages.

Parameterized alarm configurations can be created using the **pmieconf** facilities; see the **pmieconf(1)** and **pmie(1)** man pages.

# Chapter 3. PMAPI--The Performance Metrics API

## Table of Contents



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."

# 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 the section called "Domains, Metrics, Instances and Labels".

For brevity and efficiency, internally PCP avoids using 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 the section called "Metrics".

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*.

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

# 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, “Metrics Sharing the Same Instance Domain”:

### Example 3.1. Metrics Sharing the Same Instance Domain

```
$ pminfo -f filesys.free

filesys.free
   inst [1 or “/dev/disk0”] value 1803
   inst [2 or “/dev/disk1”] value 22140
   inst [3 or “/dev/disk2”] value 157938
```

The metric `filesys.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/disk0`, `/dev/disk1`, and `/dev/disk2`. These instances form an instance domain that is shared by the performance metrics `filesys.capacity`, `filesys.used`, `filesys.free`, `filesys.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.cpu.sys` 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.rss` (one instance per process), `swap.free` (one instance per swap partition) and `kernel.percpu.cpu.intr` (one instance per CPU).

# 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 a set of archive logs 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. This function returns an opaque handle which can be used to identify the context.

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 set of archive logs 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.

The current context can be changed by passing a context handle to **pmUseContext**. If a live context connection fails, the **pmReconnectContext** function can be used to attempt to reconnect it.

# Performance Metric Descriptions

For each defined performance metric, there exists metadata describing it.

- A performance metric description (**pmDesc** structure) that describes the format and semantics of the performance metric.
- Help text associated with the metric and any associated instance domain.
- Performance metric labels (name:value pairs in **pmLabelSet** structures) associated with the metric and any associated instances.

The **pmDesc** structure, in Example 3.2, " **pmDesc** Structure", 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_EVENT     9    /* packed pmEventArray */
#define PM_TYPE_UNKNOWN   255  /* used in pmValueBlock not pmDesc */
```

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

Event records are encoded as a packed array of strongly-typed, well-defined records within a `pmResult` structure, using a container metric with a value of type `PM_TYPE_EVENT`.

If the value of a performance metric is of type `PM_TYPE_STRING`, `PM_TYPE_AGGREGATE`, `PM_TYPE_AGGREGATE_STATIC`, or `PM_TYPE_EVENT`, the interpretation of that value is unknown to many PCP components. In the case of the aggregate types, 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. Strings can be manipulated using the standard C libraries. Event records contain timestamps, event flags and event parameters, and the PMAPI provides support for unpacking an event record - see the **pmUnpackEventRecords(3)** man page for details. Further discussion on event metrics and event records can be found in the section called “Performance Event Metrics”.

`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  /* instantaneous value continuous domain */
#define PM_SEM_DISCRETE 4  /* instantaneous 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 $Space^{1}.Time^{-1}$, while the running total of system calls is $Count^{1}$, memory allocation is $Space^{1}$, and average service time per event is $Time^{1}.Count^{-1}$.

- 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 $Space^{1}.Time^{-1}$ may have values encoded using the scale megabytes per second.

This information is encoded in the `pmUnits` data structure, shown in Example 3.3, “ `pmUnits` and `pmDesc` Structures”. It is embedded in the `pmDesc` structure :

The structures are as follows:

### 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 pad:8;
    int scaleCount:4; /* one of PM_COUNT_* below */
    int scaleTime:4;  /* one of PM_TIME_* below */
    int scaleSpace:4; /* one of PM_SPACE_* below */
    int dimCount:4;   /* event dimension */
    int dimTime:4;    /* time dimension */
    int dimSpace:4;   /* space dimension
} pmUnits;   /* dimensional units and scale of value */
/* pmUnits.scaleSpace */
#define PM_SPACE_BYTE 0  /* bytes */
#define PM_SPACE_KBYTE 1 /* kibibytes (1024) */
#define PM_SPACE_MBYTE 2 /* mebibytes (1024^2) */
#define PM_SPACE_GBYTE 3 /* gibibytes (1024^3) */
#define PM_SPACE_TBYTE 4 /* tebibytes (1024^4) */
#define PM_SPACE_PBYTE 5 /* pebibytes (1024^5) */
#define PM_SPACE_EBYTE 6 /* exbibytes (1024^6) */
#define PM_SPACE_ZBYTE 7 /* zebibytes (1024^7) */
#define PM_SPACE_YBYTE 8 /* yobibytes (1024^8) */
/* 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 */
```

Metric and instance domain help text are simple ASCII strings. As a result, there are no special data structures associated with them. There are two forms of help text available for each metric and instance domain, however - one-line and long form.

### Example 3.4. Help Text Flags

```
#define PM_TEXT_ONELINE 1
#define PM_TEXT_HELP    2
```

Labels are stored and communicated within PCP using JSONB formatted strings in the `json` field of a `pmLabelSet` structure. This format is a restricted form of JSON suitable for indexing and other operations. In JSONB form, insignificant whitespace is discarded, and order of label names is not

preserved. Within the PMCS, however, a lexicographically sorted key space is always maintained. Duplicate label names are not permitted. The label with highest precedence in the label hierarchy (context level labels, domain level labels, and so on) is the only one presented.

### Example 3.5. `pmLabel` and `pmLabelSet` Structures

```
typedef struct {
    uint     name : 16;      /* label name offset in JSONB string */
    uint     namelen : 8;    /* length of name excluding the null */
    uint     flags : 8;      /* information about this label */
    uint     value : 16;     /* offset of the label value */
    uint     valuelen : 16;  /* length of value in bytes */
} pmLabel;

/* flags identifying label hierarchy classes. */
#define PM_LABEL_CONTEXT        (1<<0)
#define PM_LABEL_DOMAIN         (1<<1)
#define PM_LABEL_INDOM          (1<<2)
#define PM_LABEL_CLUSTER        (1<<3)
#define PM_LABEL_ITEM           (1<<4)
#define PM_LABEL_INSTANCES      (1<<5)
/* flag identifying extrinsic labels. */
#define PM_LABEL_OPTIONAL       (1<<7)

typedef struct {
    uint     inst;          /* PM_IN_NULL or the instance ID */
    int      nlabels;       /* count of labels or error code */
    char     *json;         /* JSONB formatted labels string */
    uint     jsonlen : 16;  /* JSON string length byte count */
    uint     padding : 16;  /* zero, reserved for future use */
    pmLabel  *labels;       /* indexing into the JSON string */
} pmLabelSet;
```

The `pmLabel` `labels` array provides name and value indexes and lengths in the json string.

The `flags` field is a bitfield identifying the hierarchy level and whether this name:value pair is intrinsic (optional) or extrinsic (part of the mandatory, identifying metadata for the metric or instance). All other fields are offsets and lengths in the JSONB string from an associated `pmLabelSet` structure.

# Performance Metrics Values

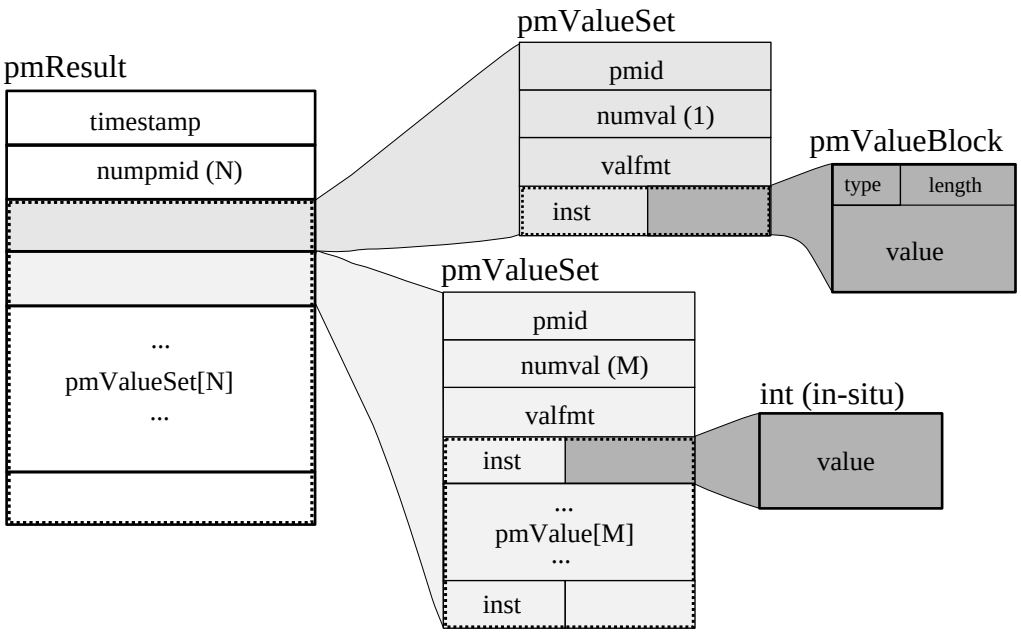
An application may fetch (or store) values for a set of performance metrics, each with a set of associated instances, using a single **pmFetch** (or **pmStore**) function call. To accommodate this, values are delivered across the PMAPI in the form of a tree data structure, rooted at a `pmResult` structure. This encoding is illustrated in Figure 3.1, “A Structured Result for Performance Metrics from **pmFetch**”, and uses the component data structures in Example 3.6, “ **pmValueBlock** and **pmValue** Structures”:

### Example 3.6. 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 3.7, "`pmValueBlock` Structure", and will be located via `pval`:

The `pmValueBlock` structure is as follows:

### Example 3.7. `pmValueBlock` Structure

```
typedef struct {
    unsigned int    vlen : 24;    /* bytes for vtype/vlen + vbuf */
    unsigned int    vtype : 8;    /* value type */
    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 `<pcp/pmapi.h>`.

A `pmValueSet` structure, as shown in Example 3.8, "`pmValueSet` Structure", contains all of the values to be returned from **pmFetch** for a single performance metric identified by the `pmid` field.

### Example 3.8. `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 3.9, " `pmResult` Structure", contains a time stamp and an array of `numpmid` pointers to `pmValueSet`.

### Example 3.9. `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 metric values were extracted from their respective domains.

#### 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 the section called "PMAPI Ancillary Support Services".

# Performance Event Metrics

In addition to performance metric values which are sampled by monitor tools, PCP supports the notion of performance event metrics which occur independently to any sampling frequency. These event metrics (PM_TYPE_EVENT) are delivered using a novel approach which allows both sampled and event trace data to be delivered via the same live wire protocol, the same on-disk archive format, and fundamentally

using the same PMAPI services. In other words, a monitor tool may be sample and trace, simultaneously, using the PMAPI services discussed here.

Event metrics are characterised by certain key properties, distinguishing them from the other metric types (counters, instantaneous, and discrete):

- Occur at times outside of any monitor tools control, and often have a fine-grained timestamp associated with each event.
- Often have parameters associated with the event, which further describe each individual event, as shown in Figure 3.2, “Sample **write(2)** syscall entry point encoding”.
- May occur in very rapid succession, at rates such that both the collector and monitor sides may not be able to track all events. This property requires the PCP protocol to support the notion of "dropped" or "missed" events.
- There may be inherent relationships between events, for example the start and commit (or rollback) of a database transaction could be separate events, linked by a common transaction identifier (which would likely also be one of the parameters to each event). Begin-end and parent-child relationships are relatively common, and these properties require the PCP protocol to support the notion of "flags" that can be associated with events.

These differences aside, the representation of event metrics within PCP shares many aspects of the other metric types - event metrics appear in the Name Space (as do each of the event parameters), each has an associated Performance Metric Identifier and Descriptor, may have an instance domain associated with them, and may be recorded by **pmlogger** for subsequent replay.

### Figure 3.2. Sample write(2) syscall entry point encoding

```
write ( 7, "It was the best of times, ...", 4096 );
```

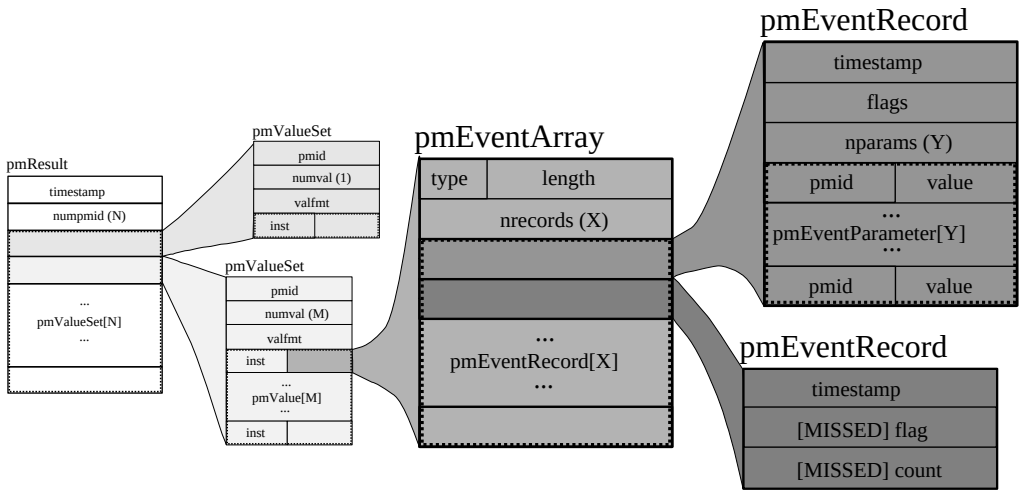
PCP Metrics:

| | |
|---|---|
| event.syscall.write_entry | (PM_TYPE_EVENT) |
| event.syscall.params.fd | (PM_TYPE_32) |
| event.syscall.params.user_buffer | (PM_TYPE_AGGREGATE) |
| event.syscall.params.buffer_size | (PM_TYPE_64, PM_SPACE_BYTE) |
| event.syscall.params.pid | (PM_TYPE_32) |

Event metrics and their associated information (parameters, timestamps, flags, and so on) are delivered to monitoring tools alongside sampled metrics as part of the **pmResult** structure seen previously in Example 3.9, “ `pmResult` Structure”.

The semantics of **pmFetch(3)** specifying an event metric PMID are such that all events observed on the collector since the previous fetch (by this specific monitor client) are to transfered to the monitor. Each event will have the metadata described earlier encoded with it (timestamps, flags, and so on) for each event. The encoding of the series of events involves a compound data structure within the **pmValueSet** associated with the event metric PMID, as illustrated in Figure 3.3, “Result Format for Event Performance Metrics from **pmFetch**”.

**Figure 3.3. Result Format for Event Performance Metrics from pmFetch**




At the highest level, the "series of events" is encapsulated within a **pmEventArray** structure, as in Example 3.10, " **pmEventArray** and **pmEventRecord** Structures":

### Example 3.10. pmEventArray and pmEventRecord Structures

```
typedef struct {
    pmTimeval      er_timestamp;  /* 2 x 32-bit timestamp format */
    unsigned int     er_flags;      /* event record characteristics */
    int              er_nparams;    /* number of ea_param[] entries */
    pmEventParameter er_param[1];
} pmEventRecord;

typedef struct {
    unsigned int     ea_len :  24;  /* bytes for type/len + records */
    unsigned int     ea_type : 8;   /* value type */
    int              ea_nrecords;   /* number of ea_record entries */
    pmEventRecord    ea_record[1];
} pmEventArray;
```

Note that in the case of dropped events, the **pmEventRecord** structure is used to convey the number of events dropped - *er_flags* is used to indicate the presence of dropped events, and *er_nparams* is used to hold a count. Unsurprisingly, the parameters (*er_param*) will be empty in this situation.

The pmEventParameter structure is as follows:

### Example 3.11. pmEventParameter Structure

```
typedef struct {
    pmID             ep_pmid;       /* parameter identifier */
    unsigned int     ep_type;       /* value type */
    int              ep_len;        /* bytes for type/len + vbuf */
    /* actual value (vbuf) here */
} pmEventParameter;
```

# Event Monitor Considerations

In order to simplify the decoding of event record arrays, the PMAPI provides the **pmUnpackEventRecords** function for monitor tools. This function is passed a pointer to a **pmValueSet** associated with an event metric (within a **pmResult**) from a **pmFetch(3)**. For a given instance of that event metric, it returns an array of "unpacked" **pmResult** structures for each event.

The control information (flags and optionally dropped events) is included as derived metrics within each result structure. As such, these values can be queried similarly to other metrics, using their names - `event.flags` and `event.missed`. Note that these metrics will only exist after the first call to **pmUnpackEventRecords**.

An example of decoding event metrics in this way is presented in Example 3.12, “Unpacking Event Records from an Event Metric `pmValueSet`”:

### Example 3.12. Unpacking Event Records from an Event Metric `pmValueSet`

```
enum { event_flags = 0, event_missed = 1 };
static char *metadata[] = { "event.flags", "event.missed" };
static pmID metapmid[2];

void dump_event(pmValueSet *vsp, int idx)
{
    pmResult    **res;
    int         r, sts, nrecords;

    nrecords = pmUnpackEventRecords(vsp, idx, &res);
    if (nrecords < 0)
        fprintf(stderr, " pmUnpackEventRecords: %s\n", pmErrStr(nrecords));
    else
        printf(" %d event records\n", nrecords);

    if ((sts = pmLookupName(2, &metadata, &metapmid)) < 0) {
        fprintf(stderr, "Event metadata error: %s\n", pmErrStr(sts));
        exit(1);
    }

    for (r = 0; r < nrecords; r++)
        dump_event_record(res, r);

    if (nrecords >= 0)
        pmFreeEventResult(res);
}

void dump_event_record(pmResult *res, int r)
{
    int         p;

    pmPrintStamp(stdout, &res[r]->timestamp);
    if (res[r]->numpmid == 0)
        printf(" ==> No parameters\n");
    for (p = 0; p < res[r]->numpmid; p++) {
        pmValueSet  *vsp = res[r]->vset[p];
```

```
        if (vsp->numval < 0) {
            int error = vsp->numval;
            printf("%s: %s\n", pmIDStr(vsp->pmid), pmErrStr(error));
        } else if (vsp->pmid == metapmid[event_flags]) {
            int flags = vsp->vlist[0].value.lval;
            printf(" flags 0x%x (%s)\n", flags, pmEventFlagsStr(flags));
        } else if (vsp->pmid == metapmid[event_missed]) {
            int count = vsp->vlist[0].value.lval;
            printf(" ==> %d missed event records\n", count);
        } else {
            dump_event_record_parameters(vsp);
        }
    }
}

void dump_event_record_parameters(pmValueSet *vsp)
{
    pmDesc      desc;
    char        *name;
    int         sts, j;

    if ((sts = pmLookupDesc(vsp->pmid, &desc)) < 0) {
        fprintf(stderr, "pmLookupDesc: %s\n", pmErrStr(sts));
    } else
    if ((sts = pmNameID(vsp->pmid, &name)) < 0) {
        fprintf(stderr, "pmNameID: %s\n", pmErrStr(sts));
    } else {
        printf("parameter %s", name);
        for (j = 0; j < vsp->numval; j++) {
            pmValue *vp = &vsp->vlist[j];
            if (vsp->numval > 1) {
                printf("[%d]", vp->inst);
                pmPrintValue(stdout, vsp->valfmt, desc.type, vp, 1);
                putchar('\n');
            }
        }
        free(name);
    }
}
```

# Event Collector Considerations

There is a feedback mechanism that is inherent in the design of the PCP monitor-collector event metric value exchange, which protects both monitor and collector components from becoming overrun by high frequency event arrivals. It is important that PMDA developers are aware of this mechanism and all of its implications.

Monitor tools can query new event arrival on whatever schedule they choose. There are no guarantees that this is a fixed interval, and no way for the PMDA to attempt to dictate this interval (nor should there be).

As a result, a PMDA that provides event metrics must:

- Track individual client connections using the per-client PMDA extensions (PMDA_INTERFACE_5 or later).

- Queue events, preferably in a memory-efficient manner, such that each interested monitor tool (there may be more than one) is informed of those events that arrived since their last request.
- Control the memory allocated to in-memory event storage. If monitors are requesting new events too slowly, compared to event arrival on the collector, the "missed events" feedback mechanism must be used to inform the monitor. This mechanism is also part of the model by which a PMDA can fix the amount of memory it uses. Once a fixed space is consumed, events can be dropped from the tail of the queue for each client, provided a counter is incremented and the client is subsequently informed.

### Note

It is important that PMDAs are part of the performance solution, and not part of the performance problem! With event metrics, this is much more difficult to achieve than with counters or other sampled values.

There is certainly elegance to this approach for event metrics, and the way they dovetail with other, sampled performance metrics is unique to PCP. Notice also how the scheme naturally allows multiple monitor tools to consume the same events, no matter what the source of events is. The downside to this flexibility is increased complexity in the PMDA when event metrics are used.

This complexity comes in the form of event queueing and memory management, as well as per-client state tracking. Routines are available as part of the **pcp_pmda** library to assist, however - refer to the man page entries for **pmdaEventNewQueue(3)** and **pmdaEventNewClient(3)** for further details.

One final set of helper APIs is available to PMDA developers who incorporate event metrics. There is a need to build the **pmEventArray** structure, introduced in Example 3.10, “ **pmEventArray** and **pmEventRecord** Structures”. This can be done directly, or using the helper routine **pmdaEventNewArray(3)**. If the latter, simpler model is chosen, the closely related routines **pmdaEventAddRecord**, **pmdaEventAddParam** and **pmdaEventAddMissedRecord** would also usually be used.

Depending on the nature of the events being exported by a PMDA, it can be desirable to perform **filtering** of events on the collector system. This reduces the amount of event traffic between monitor and collector systems (which may be filtered further on the monitor system, before presenting results). Some PMDAs have had success using the **pmStore(3)** mechanism to allow monitor tools to send a filter to the PMDA - using either a special control metric for the store operation, or the event metric itself. The filter sent will depend on the event metric, but it might be a regular expression, or a tracing script, or something else.

This technique has also been used to **enable** and **disable** event tracing entirely. It is often appropriate to make use of authentication and user credentials when providing such a facility (PMDA_INTERFACE_6 or later).

# PMAPI Programming Style and Interaction

The following sections describe the PMAPI programming style:

- Variable length argument and results lists
- Python specific issues
- PMAPI error handling

## 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**, **pmLookupLabels** 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. Similarly, the **pmLookupLabels** function has an associated function (**pmFreeLabelSets**) to release the storage.

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**, **pmFreeLabelSets**, or **pmFreeResult** should not be called.

# Python Specific Issues

A pcp client may be written in the python language by making use of the python bindings for PMAPI. The bindings use the python ctypes module to provide an interface to the PMAPI C language data structures. The primary imports that are needed by a client are:

- cpmapi which provides access to PMAPI constants

```
import cpmapi as c_api
```

- pmapi which provides access to PMAPI functions and data structures

```
from pcp import pmapi
```

- pmErr which provides access to the python bindings exception handler

```
from pcp.pmapi import pmErr
```

- pmgui which provides access to PMAPI record mode functions

```
from pcp import ppmgui
```

Creating and destroying a PMAPI context in the python environment is done by creating and destroying an object of the pmapi class. This is done in one of two ways, either directly:

```
context = pmapi.pmContext()
```

or by automated processing of the command line arguments (refer to the **pmGetOptions** man page for greater detail).

```
options = pmapi.pmOptions(...)
context = pmapi.pmContext.fromOptions(options, sys.argv)
```

Most PMAPI C functions have python equivalents with similar, although not identical, call signatures. Some of the python functions do not return native python types, but instead return native C types wrapped by the ctypes library. In most cases these types are opaque, or nearly so; for example *`pmid`*:

```
pmid = context.pmLookupName("mem.freemem")
```

```
desc = context.pmLookupDescs(pmid)
result = context.pmFetch(pmid)
...
```

See the comparison of a standalone C and python client application in Example 3.25, “PMAPI Error Handling”.

# 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)** and **pmErrStr_r(3)** man pages. The error condition is returned as the function value from a previous PMAPI call; there is no global error indicator (unlike `errno`). This is to accommodate multi-threaded performance tools.

The available error codes may be displayed with the following command:

```
pmerr -l
```

Where possible, PMAPI routines are made as tolerant to failure as possible. In particular, routines which deal with compound data structures - results structures, multiple name lookups in one call and so on, will attempt to return all data that can be returned successfully, and errors embedded in the result where there were (partial) failures. In such cases a negative failure return code from the routine indicates catastrophic failure, otherwise success is returned and indicators for the partial failures are returned embedded in the results.

# 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 fetchgroup services
- PMAPI record-mode services
- PMAPI archive-specific services

- PMAPI time control services
- PMAPI ancillary support services

# PMAPI Name Space Services

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

## pmGetChildren Function

```
int pmGetChildren(const char*name, char***offspring)
Python:
[name1, name2...] = pmGetChildren(name)
```

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.

For the python bindings a tuple containing the relative names of all the immediate descendents of *name* in the current PMNS is 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).

The python bindings return a tuple containing the relative names of all the immediate descendents of *name*, where *name* is a full pathname to a node in the current PMNS.

## pmGetChildrenStatus Function

```
int pmGetChildrenStatus(const char *name, char ***offspring, int **status)
Python:
([name1, name2...],[status1, status2...]) = pmGetChildrenStatus(name)
```

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.

The python bindings return a tuple containing the relative names and statuses of all the immediate descendents of *name*, where *name* is a full pathname to a node in the current PMNS.

## pmGetPMNSLocation Function

```
int pmGetPMNSLocation(void)
Python:
int loc = pmGetPMNSLocation()
```

If an application needs to know where the origin of a PMNS is, **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.

The python bindings return whether a PMNS is an archive *cpmapi.PMNS_ARCHIVE*, a local PMNS file *cpmapi.PMNS_LOCAL*, or a remote PMCD *cpmapi.PMNS_REMOTE*. The constants are available by importing cpmapi.

## pmLoadNameSpace Function

```
int pmLoadNameSpace(const char *filename)
Python:
int status = pmLoadNameSpace(filename)
```

In the highly unusual situation that an application wants to force using a local Performance Metrics Name Space (PMNS), the application can 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 is stored in an ASCII format.

The python bindings load a local tailored Name Space from *filename*.

### Note

Do not use this routine in monitor tools. The distributed PMNS services avoid the need for a local PMNS; so applications should **not** use **pmLoadNameSpace**. Without this call, the default PMNS is the one at the source of the performance metrics (PMCD or an archive).

## pmLookupName Function

```
int pmLookupName(int numpmid, char *namelist[], pmID pmidlist[])
Python:
c_uint pmid [] = pmLookupName("MetricName")
c_uint pmid [] = pmLookupName(("MetricName1", "MetricName2", ...))
```

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.

The python bindings return an array of associated PMIDs corresponding to a tuple of *MetricNames*. The returned *pmid* tuple is passed to **pmLookupDescs** and **pmFetch**.

## pmNameAll Function

```
int pmNameAll(pmID pmid, char ***nameset)
Python:
[name1, name2...] = pmNameAll(pmid)
```

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.

The python bindings return a tuple of all metric names having this identical *pmid*.

## pmNameID Function

```
int pmNameID(pmID pmid, char **name)
Python:
"metric name" = pmNameID(pmid)
```

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.

The python bindings return a metric name corresponding to a *pmid*.

## pmTraversePMNS Function

```
int pmTraversePMNS(const char *name, void (*dometric)(const char *))
Python:
int status = pmTraversePMNS(name, traverse_callback)
```

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(3)** man page.

The python bindings perform a depth first traversal of the PMNS by scanning *namespace*, depth first, and call a python function *traverse_callback* for each node.

## pmUnloadNameSpace Function

```
int pmUnloadNameSpace(void)
Python:
pmUnLoadNameSpace("NameSpace")
```

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.

As discussed in the section called " **pmLoadNameSpace** Function" there are few if any situations where clients need to call this routine in modern versions of PCP.

# PMAPI Metrics Description Services

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

## pmLookupDesc Function

```
int pmLookupDesc(pmID pmid, pmDesc *desc)
Python:
pmDesc* pmdesc = pmLookupDesc(c_uint pmid)
(pmDesc* pmdesc)[] = pmLookupDescs(c_uint pmids[N])
(pmDesc* pmdesc)[] = pmLookupDescs(c_uint pmid)
```

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 the section called "Performance Metric Descriptions".

The python bindings return the metric description structure pmDesc corresponding to *pmid*. The returned *pmdesc* is passed to **pmExtractValue** and **pmLookupInDom**. The python bindings provide an entry **pmLookupDescs** that is similar to pmLookupDesc but does a metric description lookup for each element in a PMID array *pmids*.

## pmLookupInDomText Function

```
int pmLookupInDomText(pmInDom indom, int level, char **buffer)
Python:
"metric description" = pmGetInDomText(pmDesc pmdesc)
```

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(3)** and **free(3)** man pages.

The help text files used to implement `pmLookupInDomText` are often created using **newhelp** and accessed by the appropriate PMDA response to requests forwarded to the PMDA by PMCD. Further details may be found in the section called “PMDA Help Text”.

The python bindings lookup the description text about the performance metrics pmDesc *`pmdesc`*. The default is a one line summary; for a more verbose description add an optional second parameter *`cpmapi.PM_TEXT_HELP`*. The constant is available by importing cpmapi.

## pmLookupText Function

```
int pmLookupText(pmID pmid, int level, char **buffer)
Python:
"metric description" = pmLookupText(c_uint pmid)
```

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(3)** and **free(3)** 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 the section called “PMDA Help Text”.

The python bindings lookup the description text about the performance metrics pmID *`pmid`*. The default is a one line summary; for a more verbose description add an optional second parameter *`cpmapi.PM_TEXT_HELP`*. The constant is available by importing cpmapi.

## pmLookupLabels Function

```
int pmLookupLabels(pmID pmid, pmLabelSet **labelsets)
Python:
(pmLabelSet* pmlabelset)[] pmLookupLabels(c_uint pmid)
```

Retrieve `name:value` pairs providing additional identity and descriptive metadata about the performance metric identified by *`pmid`*.

The space pointed to by *`labelsets`* is allocated in **pmLookupLabels** with potentially multiple calls to **malloc** and it is the responsibility of the caller to **pmFreeLabelSets** the space when it is no longer required; see the **malloc(3)** and **pmFreeLabelSets(3)** man pages.

Additional helper interfaces are also available, used internally by **pmLookupLabels** and to help with post-processing of *`labelsets`*. See the **pmLookupLabels(3)** and **pmMergeLabelSets(3)** man pages.

# PMAPI Instance Domain Services

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

## pmGetInDom Function

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

```
([instance1, instance2...] [name1, name2...]) pmGetInDom(pmDesc pmdesc)
```

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(3)** and **free(3)** man pages.

The python bindings return a tuple of the instance identifiers and instance names for an instance domain *pmdesc*.

## pmLookupInDom Function

```
int pmLookupInDom(pmInDom indom, const char *name)
Python:
int instid = pmLookupInDom(pmDesc pmdesc, "Instance")
```

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.

The python bindings return the instance id corresponding to *"Instance"* in the instance domain *pmdesc*.

## pmNameInDom Function

```
int pmNameInDom(pmInDom indom, int inst, char **name)
Python:
"instance id" = pmNameInDom(pmDesc pmdesc, c_uint instid)
```

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(3)** and **free(3)** man pages.

The python bindings return the text name of an instance corresponding to an instance domain *pmdesc* with instance identifier *instid*.

# PMAPI Context Services

Table 3.1, “Context Components of PMAPI Functions ” 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 |
|---|---|---|---|---|
| **pmAddProfile** | Yes | Yes | | |

| Function Name | Metrics Source | Instance Profile | Collection Time | Notes |
|---|---|---|---|---|
| **pmDelProfile** | Yes | Yes | | |
| **pmDupContext** | Yes | Yes | Yes | |
| **pmFetch** | Yes | Yes | Yes | |
| **pmFetchArchive** | Yes | | Yes | (1) |
| **pmGetArchiveEnd** | Yes | | | (1) |
| **pmGetArchiveLabel** | Yes | | | (1) |
| **pmGetChildren** | Yes | | | |
| **pmGetChildrenStatus** | Yes | | | |
| **pmGetContextHostName** | Yes | | | |
| **pmGetPMNSLocation** | Yes | | | |
| **pmGetInDom** | Yes | | Yes | (2) |
| **pmGetInDomArchive** | Yes | | | (1) |
| **pmLookupDesc** | Yes | | | (3) |
| **pmLookupInDom** | Yes | | Yes | (2) |
| **pmLookupInDomArchive** | Yes | | | (1,2) |
| **pmLookupInDomText** | Yes | | | |
| **pmLookupLabels** | Yes | | | |
| **pmLookupName** | Yes | | | |
| **pmLookupText** | Yes | | | |
| **pmNameAll** | Yes | | | |
| **pmNameID** | Yes | | | |
| **pmNameInDom** | Yes | | Yes | (2) |
| **pmNameInDomArchive** | Yes | | | (1,2) |
| **pmSetMode** | Yes | | Yes | |
| **pmStore** | Yes | | | (4) |
| **pmTraversePMNS** | 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. 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.

## pmNewContext Function

```
int pmNewContext(int type, const 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 specification (*type* is PM_CONTEXT_HOST) or a comma-separated list of names referring to a set of archive logs (*type* is PM_CONTEXT_ARCHIVE). Each element of the list may either be the base name common to all of the physical files of an archive log or the name of a directory containing archive logs.

A host specification usually contains a simple hostname, an internet address (IPv4 or IPv6), or the path to the PMCD Unix domain socket. It can also specify properties of the connection to PMCD, such as the protocol to use (secure and encrypted, or native) and whether PMCD should be reached via a **pmproxy** host. Various other connection attributes, such as authentication information (user name, password, authentication method, and so on) can also be specified. Further details can be found in the **PCPIntro(3)** man page, and the companion *Performance Co-Pilot Tutorials and Case Studies* document.

In the case where *type* is PM_CONTEXT_ARCHIVE, there are some restrictions on the archives within the specified set:

- The archives must all have been generated on the same host.
- The archives must not overlap in time.
- The archives must all have been created using the same time zone.
- The pmID of each metric should be the same in all of the archives. Multiple pmIDs are currently tolerated by using the first pmID defined for each metric and ignoring subsequent pmIDs.
- The type of each metric must be the same in all of the archives.
- The semantics of each metric must be the same in all of the archives.
- The units of each metric must be the same in all of the archives.
- The instance domain of each metric must be the same in all of the archives.

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 DSO PMDAs listed in the **pmcd** configuration file ${PCP_PMCDCONF_PATH}. The reason this is done, as opposed to all of the DSO PMDAs found below ${PCP_PMDAS_DIR} for example, is that DSO PMDAs listed there are very likely to have their metric names reflected in the local Name Space file, which will be loaded for this class of context.

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 first log for a set of archives. In the case of archives, the initial collection time results in the earliest set of metrics being returned from the set of archives 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**.

For the python bindings creating and destroying a PMAPI context is done by creating and destroying an object of the pmapi class.

## 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.

For the python bindings creating and destroying a PMAPI context is done by creating and destroying an object of the pmapi class.

## 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.

## 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.

## pmWhichContext Function

```
int pmWhichContext(void)
Python:
int ctx_idx = pmWhichContext()
```

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

The python bindings return the handle of the current PMAPI context.

## pmAddProfile Function

```
int pmAddProfile(pmInDom indom, int numinst, int instlist[])
Python:
int status = pmAddProfile(pmDesc pmdesc, [c_uint instid])
```

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).
```

The python bindings add the list of instances *instid* to the instance profile of the instance *pmdesc*.

## pmDelProfile Function

```
int pmDelProfile(pmInDom indom, int numinst, int instlist[])
Python:
int status = pmDelProfile(pmDesc pmdesc, c_uint instid)
int status = pmDelProfile(pmDesc pmdesc, [c_uint instid])
```

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)
```

The python bindings delete the list of instances *instid* from the instance profile of the instance domain *pmdesc*.

## pmSetMode Function

```
int pmSetMode(int mode, const struct timeval *when, int delta)
Python:
int status = pmSetMode(mode, timeVal timeval, 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 set of PCP archive logs by alternating calls to **pmSetMode** and **pmFetch**.

In Example 3.13, “Dumping Values in Temporal Sequence”, 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 3.13. 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 3.14, “Replaying Interpolated Metrics” may be used to replay interpolated metrics from an archive in reverse chronological order, at ten-second intervals (of recorded time):

### Example 3.14. 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);
}
```

The python bindings define the collection *time* and *mode* for reading archive files. *mode* can be one of: c_api.PM_MODE_LIVE, c_api.PM_MODE_INTERP, c_api.FORW, c_api.BACK. wjocj are available by importing cpmapi.

## pmReconnectContext Function

```
int pmReconnectContext(int handle)
Python:
int status = pmReconnectContext()
```

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 retry 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.

The python bindings reestablish the connection for the context.

## pmGetContextHostName Function

```
const char *pmGetContextHostName(int id)
char *pmGetContextHostName_r(int id, char *buf, int buflen)
Python:
"hostname" = pmGetContextHostName()
```

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.

This function will use the pmcd.hostname metric if it is available, and so is able to provide an accurate hostname in the presence of connection tunnelling and port forwarding.

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

In the case of **pmGetContextHostName**, the string value is held in a single static buffer, so concurrent calls may not produce the desired results. The **pmGetContextHostName_r** function allows a buffer and length to be passed in, into which the message is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings query the current context hostname.

# PMAPI Timezone Services

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

## pmNewContextZone Function

```
int pmNewContextZone(void)
Python:
pmNewContextZone()
```

If the current PMAPI context is an archive, the **pmNewContextZone** function uses the timezone from the archive label record in the first archive of the set 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**.

## pmNewZone Function

```
int pmNewZone(const char *tz)
Python:
int tz_handle = pmNewZone(int 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(7)** man page.

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

The python bindings create a new zone handle and set reporting timezone for the timezone defined by *tz*.

## pmUseZone Function

```
int pmUseZone(const int tz_handle)
Python:
int status = pmUseZone(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.

The python bindings set the current reporting timezone defined by timezone *tz_handle*.

## pmWhichZone Function

```
int pmWhichZone(char **tz)
Python:
"zone string" = pmWhichZone()
```

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.

The python bindings return the current reporting timezone.

# PMAPI Metrics Services

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

## pmFetch Function

```
int pmFetch(int numpmid, pmID pmidlist[], pmResult **result)
Python:
pmResult* pmresult = pmFetch(c_uint pmid[])
```

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 the section called “PMAPI Context Services”.

The principal result from **pmFetch** is returned as a tree structured *result*, described in the the section called “Performance Metrics Values”.

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 3.15, “PMAPI Metrics Services”, 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 3.15. PMAPI Metrics Services

```
int       i, j, sts;
pmID      pmidlist[10];
pmResult  *result;
time_t    now;

/* set up PMAPI context, numpmid and pmidlist[] ... */
while ((sts = pmFetch(10, pmidlist, &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.

The python bindings fetch a pmResult corresponding to a *pmid* list, which is returned from **pmLookupName**. The returned *pmresult* is passed to **pmExtractValue**.

## pmFreeResult Function

```
void pmFreeResult(pmResult *result)
Python:
pmFreeResult(pmResult* pmresult)
```

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

THe python bindings free a *pmresult* previously allocated by **pmFetch**.

## pmStore Function

```
int pmStore(const pmResult *request)
Python:
pmResult* pmresult = pmStore(pmResult* pmresult)
```

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.

# PMAPI Fetchgroup Services

The fetchgroup functions implement a registration-based mechanism to fetch groups of performance metrics, including automation for general unit, rate, type conversions and convenient instance and value encodings. They constitute a powerful and compact alternative to the classic Performance Metrics Application Programming Interface (PMAPI) sequence of separate lookup, check, fetch, iterate, extract, and convert functions.

A fetchgroup consists of a PMAPI context and a list of metrics that the application is interested in fetching. For each metric of interest, a conversion specification and a destination **pmAtomValue** pointer is given. Then, at each subsequent fetchgroup-fetch operation, all metrics are fetched, decoded/converted, and deposited in the desired field of the destination **pmAtomValue**s. See Example 3.18, “ pmAtomValue Structure” for more on that data type. Similarly, a per-metric-instance status value is optionally available for detailed diagnostics reflecting fetch/conversion.

The **pmfetchgroup(3)** man pages give detailed information on the C API; we only list some common cases here. The simplified Python binding to the same API is summarized below. One difference is that runtime errors in C are represented by status integers, but in Python are mapped to **pmErr** exceptions. Another is that supplying metric type codes are mandatory in the C API but optional in Python, since the latter language supports dynamic typing. Another difference is Python's wrapping of output metric values in callable "holder" objects. We demonstrate all of these below.

## Fetchgroup setup

To create a fetchgroup and its private PMAPI context, the **pmCreateFetchGroup** function is used, with parameters similar to **pmNewContext** (see the section called “ **pmNewContext** Function”).

```
int sts;
pmFG fg;
sts = pmCreateFetchGroup(& fg, PM_CONTEXT_ARCHIVE, "./foo.meta");
assert(sts == 0);
Python
fg = pmapi.fetchgroup(c_api.PM_CONTEXT_ARCHIVE, './foo.meta')
```

If special PMAPI query, PMNS enumeration, or configuration upon the context is needed, the private context may be carefully accessed.

```
int ctx = pmGetFetchGroupContext(fg);
sts = pmUseContext(ctx);
assert(sts == 0);
sts = pmSetMode(...);
```

**Python**

```
ctx = fg.get_context()
ctx.pmSetMode(...)
```

A fetchgroup is born empty. It needs to be extended with metrics to read. Scalars are easy. We specify the metric name, an instance-domain instance if necessary, a unit-scaling and/or rate-conversion directive if desired, and a type code (see Example 3.2, " **pmDesc** Structure"). In C, the value destination is specified by pointer. In Python, a value-holder is returned.

```
static pmAtomValue ncpu, loadavg, idle;
sts = pmExtendFetchGroup_item(fg, "hinv.ncpu", NULL, NULL,
                              & ncpu, PM_TYPE_32, NULL);
assert (sts == 0);
sts = pmExtendFetchGroup_item(fg, "kernel.all.load", "5 minute", NULL,
                              & loadavg, PM_TYPE_DOUBLE, NULL);
assert (sts == 0);
sts = pmExtendFetchGroup_item(fg, "kernel.all.cpu.idle", NULL, "s/100s",
                              & idle, PM_TYPE_STRING, NULL);
assert (sts == 0);
```

**Python**

```
ncpu = fg.extend_item('hinv.cpu')
loadavg = fg.extend_item('kernel.all.load', instance='5 minute')
idle = fg.extend_item('kernel.all.cpu.idle, scale='s/100s')
```

Registering metrics with whole instance domains are also possible; these result in a vector of **pmAtomValue** instances, instance names and codes, and status codes, so the fetchgroup functions take more optional parameters. In Python, a value-holder-iterator object is returned.

```
enum { max_disks = 100 };
static unsigned num_disks;
static pmAtomValue disk_reads[max_disks];
static int disk_read_stss[max_disks];
static char *disk_names[max_disks];
sts = pmExtendFetchGroup_indom(fg, "disk.dm.read", NULL,
                               NULL, disk_names, disk_reads, PM_TYPE_32,
                               disk_read_stss, max_disks, & num_disks,
                               NULL);
```

**Python**

```
values = fg.extend_indom('disk.dm.read')
```

Registering interest in the future fetch-operation timestamp is also possible. In python, a datetime-holder object is returned.

```
struct timeval tv;
```

```
sts = pmExtendFetchGroup_timestamp(fg, & tv);
Python
tv = fg.extend_timestamp()
```

## Fetchgroup operation

Now it's time for the program to process the metrics. In the C API, each metric value is put into status integers (if requested), and one field of the **pmAtomValue** union - whichever was requested with the **PM_TYPE_*** code. In the Python API, each metric value is accessed by calling the value-holder objects.

```
sts = pmFetchGroup(fg);
assert (sts == 0);
printf("%s", ctime(& tv.tv_sec));
printf("#cpus: %d, loadavg: %g, idle: %s\n", ncpu.l, loadavg.d, idle.cp);
for (i=0; i<num_disks; i++)
    if (disk_read_stss[i] == 0)
        printf("disk %s reads %d\n", disk_names[i], disk_reads[i].l);
Python
fg.fetch()
print(tv())
print("#cpus: %d, loadavg: %g, idle: %d\n" % (ncpu(), loadavg(), idle()))
for icode, iname, value in values():
    print('disk %s reads %d' % (iname, value()))
```

The program may fetch and process the values only once, or in a loop. The program need not - *must not* - modify or free any of the output values/pointers supplied by the fetchgroup functions.

## Fetchgroup shutdown

Should the program wish to shut down a fetchgroup explicitly, thereby closing the private PMAPI context, there is a function for that.

```
sts = pmDestroyFetchGroup(fg);
Python
del fg # or nothing
```

# PMAPI Record-Mode Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) record-mode services. These services allow a monitor tool to establish connections to **pmlogger** co-processes, which they create and control for the purposes of recording live performance data from (possibly) multiple hosts. Since **pmlogger** records for one host only, these services can administer a group of loggers, and set up archive folios to track the logs. Tools like **pmafm** can subsequently use those folios to replay recorded data with the initiating tool. **pmchart** uses these concepts when providing its Record mode functionality.

## pmRecordAddHost Function

```
int pmRecordAddHost(const char *host, int isdefault, pmRecordHost **rhp)
Python:
(int status, pmRecordHost* rhp) = pmRecordAddHost("host string", 1, "configure str
```

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 `EINVAL`.

## pmRecordControl Function

```
int pmRecordControl(pmRecordHost *rhp, int request, const char *options)
Python:
int status = pmRecordControl("host string", 1, "configure string")
```

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 `EINVAL`.

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

## pmRecordSetup Function

```
FILE *pmRecordSetup(const char *folio, const char *creator, int replay)
Python:
int status = pmRecordSetup("folio string", "creator string", 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 **pmafm** 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 3.16, "`pmRecordHost` Structure", is defined as:

### Example 3.16. `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, "Creating a Recording Session", 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, as returned `pmGetProgname()`). 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 the section called "**pmRecordAddHost** Function" 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 the section called "**pmRecordControl** Function" 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 **pmconfirm**. This default is based on the assumption that most applications launching a recording session are GUI-based. In the event that **pmconfirm** 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 **pmconfirm** 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 **mkstemp**.

- 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 **pmafm** 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.

# PMAPI Archive-Specific Services

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

## pmGetArchiveLabel Function

```
int pmGetArchiveLabel(pmLogLabel *lp)
Python:
pmLogLabel loglabel = pmGetArchiveLabel()
```

Provided the current PMAPI context is associated with a set of PCP archive logs, the **pmGetArchiveLabel** function may be used to fetch the label record from the first archive in the set of archives. The structure returned through *lp* is as shown in Example 3.17, " `pmLogLabel` Structure":

### Example 3.17. `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;
```

The python bindings get the label record from the archive.

## pmGetArchiveEnd Function

```
int pmGetArchiveEnd(struct timeval *tvp)
Python:
timeval tv = status = pmGetArchiveEnd()
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **pmGetArchiveEnd** finds the logical end of the last archive file in the set (after the last complete record in the archive), and returns the last recorded time stamp with *tvp*. This timestamp 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.

The python bindings get the last recorded timestamp from the archive.

## pmGetInDomArchive Function

```
int pmGetInDomArchive(pmInDom indom, int **instlist, char ***namelist )
Python:
((instance1, instance2...) (name1, name2...)) pmGetInDom(pmDesc pmdesc)
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **pmGetInDomArchive** scans the metadata to generate the union of all instances for the instance domain *indom* that can be found in the set of archive logs, 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(3)** and **free(3)** 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).

The python bindings return a tuple of the instance IDs and names for the union of all instances for the instance domain *pmdesc* that can be found in the archive log.

## pmLookupInDomArchive Function

```
int pmLookupInDomArchive(pmInDom indom, const char *name)
Python:
c_uint instid = pmLookupInDomArchive(pmDesc pmdesc, "Instance")
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **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.

The python bindings return the instance id in *pmdesc* corresponding to *Instance*.

## pmNameInDomArchive Function

```
int pmNameInDomArchive(pmInDom indom, int inst, char **name)
Python:
"instance id" = pmNameInDomArchive(pmDesc pmdesc, c_uint instid)
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **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(3)** and**free(3)** man pages.

The python bindings return the text name of an instance corresponding to an instance domain *pmdesc* with instance identifier *instid*.

## pmFetchArchive Function

```
int pmFetchArchive(pmResult **result)
Python:
pmResult* pmresult = pmFetchArchive()
```

This is a variant of **pmFetch** that may be used only when the current PMAPI context is associated with a set of PCP archive logs. 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 **pmlogsummary**), and the more common access to the archives would be through the **pmFetch** interface.

# 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 sets of 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**, **pmgadgets**, **pmstat**, 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, “Time Control Functions in PMAPI”:

### Table 3.2. Time Control Functions in PMAPI

| Man Page | Synopsis of Time Control Function |
|---|---|
| **pmCtime(3)** | Formats the date and time for a reporting timezone. |
| **pmLocaltime(3)** | Converts the date and time for a reporting timezone. |
| **pmParseTimeWindow(3)** | Parses time window command line arguments. |
| **pmTimeConnect(3)** | Connects to a time control server via a command socket. |
| **pmTimeDisconnect(3)** | Closes the command socket to the time control server. |
| **pmTimeGetPort(3)** | Obtains the port name of the current time control server. |
| **pmTimeRecv(3)** | Blocks until the time control server sends a command message. |
| **pmTimeSendAck(3)** | Acknowledges completion of the step command. |
| **pmTimeSendBounds(3)** | Specifies beginning and end of archive time period. |
| **pmTimeSendMode(3)** | Requests time control server to change to a new VCR mode. |
| **pmTimeSendPosition(3)** | Requests time control server to change position or update intervals. |
| **pmTimeSendTimezone(3)** | Requests time control server to change timezones. |
| **pmTimeShowDialog(3)** | Changes the visibility of the time control dialogue. |
| **pmTimeGetStatePixmap(3)** | Returns array of pixmaps representing supplied time control state. |

# 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.

## pmGetConfig Function

```
char *pmGetConfig(const char *variable)
Python:
"env variable value = pmGetConfig("env 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 **setenv** 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(5)** 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.

The python bindings return a value for environment variable *`"env variable"`* from environment or pcp config file.

## pmErrStr Function

```
const char *pmErrStr(int code)
char *pmErrStr_r(int code, char *buf, int buflen);
Python:
"error string text" = pmErrStr(int error_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 platform 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`*.

In the case of **pmErrStr**, the string value is held in a single static buffer, so concurrent calls may not produce the desired results. The **pmErrStr_r** function allows a buffer and length to be passed in, into which the message is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings return the error string corresponding to the *`error code`*.

## pmExtractValue Function

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

```
pmAtomValue atomval = 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 3.18, "`pmAtomValue` Structure":

### Example 3.18. `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, "PMAPI Type Conversion" 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**. No conversion involving the type `PM_TYPE_EVENT` is supported.

**Table 3.3. PMAPI Type Conversion**

| TYPE | 32 | U32 | 64 | U64 | FLOAT | DBLE | STRING | AGGR | EVENT |
|---|---|---|---|---|---|---|---|---|---|
| 32 | Y | S | Y | S | P | P | N | N | N |
| U32 | T | Y | Y | Y | P | P | N | N | N |
| 64 | T | T,S | Y | S | P | P | N | N | N |
| u64 | T | T | T | Y | P | P | N | N | N |
| FLOAT | P, T | P, T, S | P, T | P, T, S | Y | Y | N | N | N |
| DBLE | P, T | P, T, S | P, T | P, T, S | P | Y | N | N | N |
| STRING | N | N | N | N | N | N | Y | N | N |
| AGGR | N | N | N | N | N | N | N | Y | N |
| EVENT | N | N | N | N | N | N | N | N | N |

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(3)** and **free(3)** 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.

The python bindings extract a value from a pmValue struct *ival* stored in format *valfmt* (see **pmFetch**), and convert its type from *itype* to *otype*.

## pmConvScale Function

```
int
pmConvScale(int type, const pmAtomValue *ival, const pmUnits *iunit,
pmAtomValue *oval, pmUnits *ounit)
Python:
pmAtomValue atomval = pmConvScale(int itype, pmAtomValue value,
   pmDesc* pmdesc , int descidx, int otype)
```

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.

The python bindings convert a *value* pointed to by *pmdesc* entry *descidx* to a different scale *otype*.

## pmUnitsStr Function

```
const char *pmUnitsStr(const pmUnits *pu)
char *pmUnitsStr_r(const pmUnits *pu, char *buf, int buflen)
Python:
"units string" = pmUnitsStr(pmUnits pmunits)
```

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`.

In the case of **pmUnitsStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmUnitsStr_r** function allows a buffer and length to be passed in, into which the units are stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a pmUnits struct *pmunits* to a readable string.

## pmIDStr Function

```
const char *pmIDStr(pmID pmid)
char *pmIDStr_r(pmID pmid, char *buf, int buflen)
Python:
"ID string" = pmIDStr(int 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.

In the case of **pmIDStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmIDStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a pmID *pmid* to a readable string.

## pmInDomStr Function

```
const char *pmInDomStr(pmInDom indom)
char *pmInDomStr_r(pmInDom indom, char *buf, int buflen)
Python:
"indom" = pmGetInDom(pmDesc pmdesc)
```

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.

In the case of **pmInDomStrr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmInDomStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate an instance domain ID pointed to by a pmDesc *pmdesc* to a readable string.

## pmTypeStr Function

```
const char *pmTypeStr(int type)
char *pmTypeStr_r(int type, char *buf, int buflen)
Python:
"type" = 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.

In the case of **pmTypeStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmTypeStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a performance metric type to a readable string. Constants are available for the types, e.g. c_api.PM_TYPE_FLOAT, by importing cpmapi.

## pmAtomStr Function

```
const char *pmAtomStr(const pmAtomValue *avp, int type)
char *pmAtomStr_r(const pmAtomValue *avp, int typechar *buf, int buflen)
Python:
"value" = pmAtomStr(atom, 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.

In the case of **pmAtomStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmAtomStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a pmAtomValue *atom* having performance metric *type* to a readable string. Constants are available for the types, e.g. c_api.PM_TYPE_U32, by importing cpmapi.

## pmNumberStr Function

```
const char *pmNumberStr(double value)
char *pmNumberStr_r(double value, char *buf, int buflen)
```

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*.

## pmPrintValue Function

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

```
pmPrintValue(FILE* file, pmResult pmresult, pmdesc, vset_index, vlist_index, min_w
```

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 3.19, “Using **pmPrintValue** to Print Values” illustrates using **pmPrintValue** to print the values from a pmResult structure returned via **pmFetch**:

### Example 3.19. 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));
```

Print the value of a *pmresult* pointed to by *vset_index*/*vlist_index* and described by *pmdesc*. The format of a pmResult is described in *pmResult* The python bindings can use sys.__stdout__ as a value for *file* to display to stdout.

## pmflush Function

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

**Python:**
```
int status = pmflush()
```

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 dialogue using **pmconfirm**.

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.

## pmprintf Function

```
int pmprintf(const char *fmt, ... /*args*/);
```
**Python:**
```
pmprintf("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 **mkstemp** function to securely create a pcp-prefixed temporary file in ${PCP_TMP_DIR}. 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.

## pmSortInstances Function

```
void pmSortInstances(pmResult *result)
```
**Python:**
```
pmSortInstances (pmResult* pmresult)
```

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.

## pmParseInterval Function

```
int pmParseInterval(const char *string, struct timeval *rslt, char **errmsg)
```
**Python:**
```
(struct timeval, "error message") = pmParseInterval("time string")
```

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*.

## pmParseMetricSpec Function

```
int pmParseMetricSpec(const char *string, int isarch, char *source,
                      pmMetricSpec **rsltp, char **errmsg)
Python:
(pmMetricSpec metricspec, "error message") =
                      pmParseMetricSpec("metric specification", isarch, source)
```

The **pmParseMetricSpec** function accepts a *string* specifying the name of a PCP performance metric, and optionally the source (either a hostname, a set of PCP archive logs, or a local context) 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 3.20, "`pmMetricSpec` Structure", the `pmMetricSpec` structure, which is returned via *rsltp,* represents the parsed string.

### Example 3.20. `pmMetricSpec` Structure

```
typedef struct {
    int     isarch;      /* source type: 0 -> host, 1 -> archive, 2 -> local conte
    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 *rsltp*. 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, *rsltp* is undefined. In the case of success, *errmsg* is undefined. If *rsltp*->*ninst* is 0, then *rsltp*->*inst*[0] is undefined.

# 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 package. See the **pmclient(1)** man page, and the source code, located in `${PCP_DEMOS_DIR}/pmclient`.

# 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 **LOADAV** macro with the name **kernel.all.load**, and then be able to use **LOADAV** 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, **pmclient**:

```
          cat ${PCP_DEMOS_DIR}/pmclient/pmnsmap.spec
pmclient_init {
    hinv.ncpu   NUMCPU
}

pmclient_sample {
    kernel.all.load LOADAV
    kernel.percpu.cpu.user CPU_USR
    kernel.percpu.cpu.sys CPU_SYS
    mem.freemem  FREEMEM
    disk.all.total  DKIOPS
}
```

This **pmgenmap** input produces the C code in Example 3.21, “C Code Produced by **pmgenmap** Input”. It is suitable for including with the `#include` statement:

### Example 3.21. C Code Produced by pmgenmap Input

```
/*
 * Performance Metrics Name Space Map
 * Built by runme.sh from the file
 * pmnsmap.spec
 * on Thu Jan  9 14:13:49 EST 2014
 *
 * Do not edit this file!
 */
```

```
char *pmclient_init[] = {
#define NUMCPU 0
 "hinv.ncpu",

};


char *pmclient_sample[] = {
#define LOADAV 0
 "kernel.all.load",
#define CPU_USR 1
 "kernel.percpu.cpu.user",
#define CPU_SYS 2
 "kernel.percpu.cpu.sys",
#define FREEMEM 3
 "mem.freemem",
#define DKIOPS 4
 "disk.all.total",

};
```

# 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 3.22, “Initializing Metric Specifications”:

### Example 3.22. Initializing Metric Specifications

```
/* C code fragment from pmclient.c */
numpmid = sizeof(pmclient_sample) / sizeof(char *);
if ((pmidlist = (pmID *)malloc(numpmid * sizeof(pmidlist[0]))) == NULL) {...}
if ((sts = pmLookupName(numpmid, pmclient_sample, pmidlist)) < 0) {...}

# The equivalent python code would be
pmclient_sample = ("kernel.all.load", "kernel.percpu.cpu.user",
 "kernel.percpu.cpu.sys", "mem.freemem", "disk.all.total")
pmidlist = context.pmLookupName(pmclient_sample)
```

At this stage, **pmidlist** contains the PMID for the five metrics of interest.

# Iterative Processing of Values

Assuming the tool is required to report values every *`delta`* seconds, use code similar to that in Example 3.23, “Iterative Processing”:

### Example 3.23. Iterative Processing

```
/* censored C code fragment from pmclient.c */
while (samples == -1 || samples-- > 0) {
   if ((sts = pmFetch(numpmid, pmidlist, &crp)) < 0) { ... }
   for (i = 0; i < numpmid; i++)
      if ((sts = pmLookupDesc(pmidlist[i], &desclist[i])) < 0) { ... }
```

```
   ...
   pmExtractValue(crp->vset[FREEMEM]->valfmt, crp->vset[FREEMEM]->vlist,
                  desclist[FREEMEM].type, &tmp, PM_TYPE_FLOAT);
   pmConvScale(PM_TYPE_FLOAT, &tmp, &desclist[FREEMEM].units,
                    &atom, &mbyte_scale);
   ip->freemem = atom.f;
   ...
   __pmtimevalSleep(delta);
}

# The equivalent python code would be
FREEMEM = 3
desclist = context.pmLookupDescs(metric_names)
while (samples > 0):
    crp = context.pmFetch(metric_names)
    val = context.pmExtractValue(crp.contents.get_valfmt(FREEMEM),
                            crp.contents.get_vlist(FREEMEM, 0),
                            desclist[FREEMEM].contents.type,
                            c_api.PM_TYPE_FLOAT)
    atom = ctx.pmConvScale(c_api.PM_TYPE_FLOAT, val, desclist, FREEMEM,
          c_api.PM_SPACE_MBYTE)
    (tvdelta, errmsg) = c_api.pmParseInterval(delta)
    c_api.pmtimevalSleep(delta)
```

# Accommodating Program Evolution

The flexibility provided by the PMAPI and the **pmgenmap** utility is demonstrated by Example 3.24, “Adding a Metric”. Consider the requirement for reporting a third metric `mem.physmem`. This example shows how to add the line to the specification file:

### Example 3.24. Adding a Metric

```
mem.freemem PHYSMEM
```

Then regenerate the `#include` file, and augment pmclient.c:

```
   pmExtractValue(crp->vset[PHYSMEM]->valfmt, crp->vset[PHYSMEM]->vlist,
                  desclist[PHYSMEM].type, &tmp, PM_TYPE_FLOAT);
   pmConvScale(PM_TYPE_FLOAT, &tmp, &desclist[PHYSMEM].units,
                    &atom, &mbyte_scale);

# The equivalent python code would be:
val = context.pmExtractValue(crp.contents.get_valfmt(PHYSMEM),
   crp.contents.get_vlist(PHYSMEM, 0),
   desclist[PHYSMEM].contents.type,
   c_api.PM_TYPE_FLOAT);
```

# Handling PMAPI Errors

In Example 3.25, “PMAPI Error Handling”, the simple but complete PMAPI application demonstrates the recommended style for handling PMAPI error conditions. The python bindings use the exception mechanism to raise an exception in error cases. The python client can handle this condition by catching the `pmErr` exception. For simplicity, no command line argument processing is shown here - in practice most tools use the **pmGetOptions** helper interface to assist with initial context creation and setup.

### Example 3.25. PMAPI Error Handling

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

int
main(int argc, char* argv[])
{
    int                 sts = 0;
    char                *host = "local:";
    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;
```

```
}

# The equivalent python code would be:
import sys
import traceback
from pcp import pmapi
from cpmapi import PM_TYPE_U32

try:
    context = pmapi.pmContext()
    pmid = context.pmLookupName("mem.freemem")
    desc = context.pmLookupDescs(pmid)
    result = context.pmFetch(pmid)
    freemem = context.pmExtractValue(result.contents.get_valfmt(0),
                                     result.contents.get_vlist(0, 0),
                                     desc[0].contents.type,
                                     PM_TYPE_U32)
    print "freemem is " + str(int(freemem.ul))

except pmapi.pmErr, error:
    print "%s: %s" % (sys.argv[0], error.message())
except Exception, error:
    sys.stderr.write(str(error) + "\n")
    sys.stderr.write(traceback.format_exc() + "\n")
```

# 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/libpcp.h>` and `<pcp/pmda.h>`.

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

**cc** *mycode*.**c -lpcp**

# Chapter 4. Instrumenting Applications

## Table of Contents



This chapter provides an introduction to ways of instrumenting applications using PCP.

The first section covers the use of the Memory Mapped Value (MMV) Performance Metrics Domain Agent (PMDA) to generate customized metrics from an application. This provides a robust, extremely efficient mechanism for transferring custom instrumentation into the PCP infrastructure. It has been successfully deployed in production environments for many years, has proven immensely valuable in these situations, and can be used to instrument applications written in a number of programming languages.

The Memory Mapped Value library and PMDA is supported on every PCP platform, and is enabled by default.

### Note

A particularly expansive Java API is available from the separate Parfait [http://code.google.com/p/parfait/] project. It supports both the existing JVM instrumentation, and custom application metric extensions.

The chapter also includes information on how to use the MMV library (`libpcp_mmv`) for instrumenting an application. The example programs are installed in `${PCP_DEMOS_DIR}/mmv`.

The second section covers the design of the Trace 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.

This part of the chapter also includes information on how to use the Trace PMDA and its associated library (`libpcp_trace`) for instrumenting applications. The example programs are installed in `${PCP_DEMOS_DIR}/trace`.

### Warning

The current PCP trace library is a relatively heavy-weight solution, issuing multiple system calls per trace point, runs over a TCP/IP socket even locally and performs no event batching. As such it is not appropriate for production application instrumentation at this stage.

A revised application tracing library and PMDA are planned which will be light-weight, suitable for production system tracing, and support event metrics and other advances in end-to-end distributed application tracing.

The application instrumentation libraries are designed to encourage application developers 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.

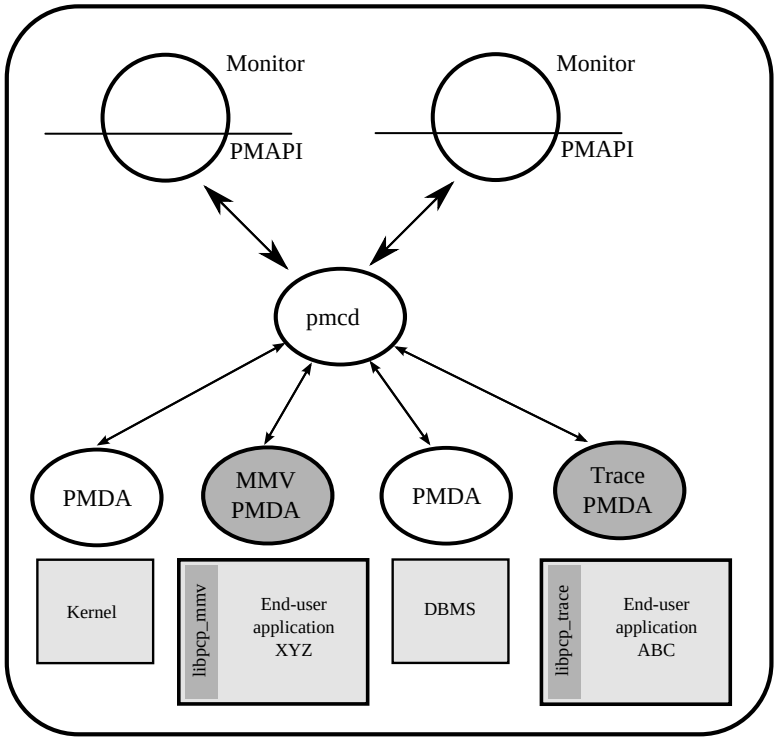
For example, developers can provide the following application performance metrics:

- 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 sub-problems solved, iteration count, transactions, data sets inspected, and so on
- Service time by operation type

# Application and Performance Co-Pilot Relationship

The relationship between an application, the `pcp_mmv` and `pcp_trace` instrumentation libraries, the MMV and Trace PMDAs, and the rest of the PCP infrastructure is shown in Figure 4.1, “Application and PCP Relationship”:

**Figure 4.1. Application and PCP Relationship**




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 of historical application performance.

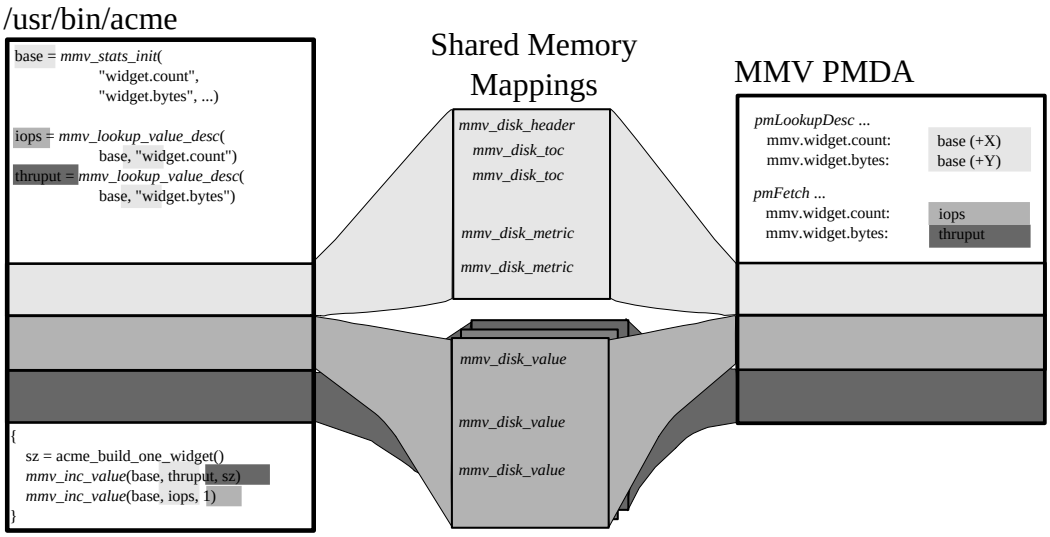
# Performance Instrumentation and Sampling

The `pcp_mmv` library provides function calls to assist with extracing important performance metrics from a program into a shared, in-memory location such that the MMV PMDA can examine and serve that information on behalf of PCP client tool requests. The `pcp_mmv` library is described in the **mmv_stats_init(3)**, **mmv_lookup_value_desc(3)**, **mmv_inc_value(3)** man pages. Additionally, the format of the shared memory mappings is described in detail in **mmv(5)**.

# MMV PMDA Design

An application instrumented with memory mapped values directly updates the memory that backs the metric values it exports. The MMV PMDA reads those values directly, from the `same` memory that the application is updating, when current values are sampled on behalf of PMAPI client tools. This relationship, and a simplified MMV API, are shown in Figure 4.2, “Memory Mapped Page Sharing”.

**Figure 4.2. Memory Mapped Page Sharing**




It is worth noting that once the metrics of an application have been registered via the `pcp_mmv` library initialisation API, subsequent interactions with the library are not intrusive to the instrumented application. At the points where values are updated, the only cost involved is the memory mapping update, which is a single memory store operation. There is no need to explicitly transfer control to the MMV PMDA, nor allocate memory, nor make system or library calls. The PMDA will only sample the values at times driven by PMAPI client tools, and this places no overhead on the instrumented application.

# Memory Mapped Values API

The `libpcp_mmv` Application Programming Interface (API) can be called from C, C++, Perl and Python (a separate project, Parfait, services the needs of Java applications). Each language has access to the complete set of functionality offered by `libpcp_mmv`. In most cases, the calling conventions differ only slightly between languages - in the case of Java and Parfait, they differ significantly however.

## Starting and Stopping Instrumentation

Instrumentation is begun with an initial call to `mmv_stats_init`, and ended with a call to `mmv_stats_stop`. These calls manipulate global state shared by the library and application. These are the only calls requiring synchonization and a single call to each is typically performed early and late in the life of the application (although they can be used to reset the library state as well, at any time). As such, the choice of synchonization primitive is left to the application, and none is currently performed by the library.

```
void *mmv_stats_init(const char *name, int cluster, mmv_stats_flags_t flags,
                     const mmv_metric_t *stats, int nstats,
                     const mmv_indom_t *indoms, int nindoms)
```

The *`name`* should be a simple symbolic name identifying the application. It is usually used as the first application-specific part of the exported metric names, as seen from the MMV PMDA. This behavior can be overriden using the *`flags`* parameter, with the MMV_FLAG_NOPREFIX flag. In the example below, full metric names such as `mmv.acme.products.count` will be created by the MMV PMDA. With the MMV_FLAG_NOPREFIX flag set, that would instead become `mmv.products.count`. It is recommended to not disable the prefix - doing so requires the applications to ensure naming conflicts do not arise in the MMV PMDA metric names.

The *cluster* identifier is used by the MMV PMDA to further distinguish different applications, and is directly used for the MMV PMDA PMID cluster field described in Example 2.3, " __pmID_int Structure", for all MMV PMDA metrics.

All remaining parameters to mmv_stats_init define the metrics and instance domains that exist within the application. These are somewhat analagous to the final parameters of pmdaInit(3), and are best explained using Example 4.1, "Memory Mapped Value Instance Structures" and Example 4.2, "Memory Mapped Value Metrics Structures". As mentioned earlier, the full source code for this example instrumented application can be found in ${PCP_DEMOS_DIR}/mmv.

### Example 4.1. Memory Mapped Value Instance Structures

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

static mmv_instances_t products[] = {
    {   .internal = 0, .external = "Anvils" },
    {   .internal = 1, .external = "Rockets" },
    {   .internal = 2, .external = "Giant_Rubber_Bands" },
};
#define ACME_PRODUCTS_INDOM 61
#define ACME_PRODUCTS_COUNT (sizeof(products)/sizeof(products[0]))

static mmv_indom_t indoms[] = {
    {   .serial = ACME_PRODUCTS_INDOM,
        .count = ACME_PRODUCTS_COUNT,
        .instances = products,
        .shorttext = "Acme products",
        .helptext = "Most popular products produced by the Acme Corporation",
    },
};
```

The above data structures initialize an instance domain of the set of products produced in a factory by the fictional "Acme Corporation". These structures are directly comparable to several concepts we have seen already (and for good reason - the MMV PMDA must interpret the applications intentions and properly export instances on its behalf):

- mmv_instances_t maps to pmdaInstid, as in Example 2.7, " pmdaInstid Structure"
- mmv_indom_t maps to pmdaIndom, as in Example 2.8, " pmdaIndom Structure" - the major difference is the addition of oneline and long help text, the purpose of which should be self-explanatory at this stage.
- *serial* numbers, as in Example 2.9, " __pmInDom_int Structure"

Next, we shall create three metrics, all of which use this instance domain. These are the mmv.acme.products metrics, and they reflect the rates at which products are built by the machines in the factory, how long these builds take for each product, and how long each product type spends queued (while waiting for factory capacity to become available).

### Example 4.2. Memory Mapped Value Metrics Structures

```
static mmv_metric_t metrics[] = {
    {   .name = "products.count",
        .item = 7,
        .type = MMV_TYPE_U64,
        .semantics = MMV_SEM_COUNTER,
```

```
        .dimension = MMV_UNITS(0,0,1,0,0,PM_COUNT_ONE),
        .indom = ACME_PRODUCTS_INDOM,
        .shorttext = "Acme factory product throughput",
        .helptext =
"Monotonic increasing counter of products produced in the Acme Corporation\n"
"factory since starting the Acme production application.  Quality guaranteed.",
    },
    {   .name = "products.time",
        .item = 8,
        .type = MMV_TYPE_U64,
        .semantics = MMV_SEM_COUNTER,
        .dimension = MMV_UNITS(0,1,0,0,PM_TIME_USEC,0),
        .indom = ACME_PRODUCTS_INDOM,
        .shorttext = "Machine time spent producing Acme products",
        .helptext =
"Machine time spent producing Acme Corporation products.  Does not include\n"
"time in queues waiting for production machinery.",
    },
    {   .name = "products.queuetime",
        .item = 10,
        .type = MMV_TYPE_U64,
        .semantics = MMV_SEM_COUNTER,
        .dimension = MMV_UNITS(0,1,0,0,PM_TIME_USEC,0),
        .indom = ACME_PRODUCTS_INDOM,
        .shorttext = "Queued time while producing Acme products",
        .helptext =
"Time spent in the queue waiting to build Acme Corporation products,\n"
"while some other Acme product was being built instead of this one.",
    },
};
#define INDOM_COUNT (sizeof(indoms)/sizeof(indoms[0]))
#define METRIC_COUNT (sizeof(metrics)/sizeof(metrics[0]))
```

As was the case with the "products" instance domain before, these metric-defining data structures are directly comparable to PMDA data structures described earlier:

- mmv_metric_t maps to a pmDesc structure, as in Example 3.2, “ **pmDesc** Structure”
- MMV_TYPE, MMV_SEM, and MMV_UNITS map to PMAPI constructs for type, semantics, dimensionality and scale, as in Example 3.3, “ `pmUnits` and `pmDesc` Structures”
- *`item`* number, as in Example 2.3, “ `__pmID_int` Structure”

For the most part, all types and macros map directly to their core PCP counterparts, which the MMV PMDA will use when exporting the metrics. One important exception is the introduction of the metric type MMV_TYPE_ELAPSED, which is discussed further in the section called “Elapsed Time Measures”.

The compound metric types - aggregate and event type metrics - are not supported by the MMV format.

# Getting a Handle on Mapped Values

Once metrics (and the instance domains they use) have been registered, the memory mapped file has been created and is ready for use. In order to be able to update the individual metric values, however, we must find get a handle to the value. This is done using the **mmv_lookup_value_desc** function, as shown in Example 4.3, “Memory Mapped Value Handles”.

**Example 4.3. Memory Mapped Value Handles**

```
#define ACME_CLUSTER 321        /* PMID cluster identifier */

int
main(int argc, char * argv[])
{
    void *base;
    pmAtomValue *count[ACME_PRODUCTS_COUNT];
    pmAtomValue *machine[ACME_PRODUCTS_COUNT];
    pmAtomValue *inqueue[ACME_PRODUCTS_COUNT];
    unsigned int working;
    unsigned int product;
    unsigned int i;

    base = mmv_stats_init("acme", ACME_CLUSTER, 0,
                          metrics, METRIC_COUNT, indoms, INDOM_COUNT);
    if (!base) {
        perror("mmv_stats_init");
        return 1;
    }

    for (i = 0; i < ACME_PRODUCTS_COUNT; i++) {
        count[i] = mmv_lookup_value_desc(base,
                        "products.count", products[i].external);
        machine[i] = mmv_lookup_value_desc(base,
                        "products.time", products[i].external);
        inqueue[i] = mmv_lookup_value_desc(base,
                        "products.queuetime", products[i].external);
    }
```

Space in the mapping file for every value is set aside at initialization time (by the **mmv_stats_init** function) - that is, space for each and every metric, and each value (instance) of each metric when an instance domain is used. To find the handle to the space set aside for one individual value requires the tuple of base memory address of the mapping, metric name, and instance name. In the case of metrics with no instance domain, the final instance name parameter should be either NULL or the empty string.

# Updating Mapped Values

At this stage we have individual handles (pointers) to each instrumentation point, we can now start modifying these values and observing changes through the PCP infrastructure. Notice that each handle is simply the canonical **pmAtomValue** pointer, as defined in Example 3.18, “ `pmAtomValue` Structure”, which is a union providing sufficient space to hold any single value.

This pointer can be either manipulated directly, or using helper functions provided by the **pcp_mmv** API, such as the **mmv_stats_inc** and **mmv_stats_set** functions.

**Example 4.4. Memory Mapped Value Updates**

```
    while (1) {
        /* choose a random number between 0-N -> product */
        product = rand() % ACME_PRODUCTS_COUNT;
```

```
        /* assign a time spent "working" on this product */
        working = rand() % 50000;

        /* pretend to "work" so process doesn't burn CPU */
        usleep(working);

        /* update the memory mapped values for this one: */
        /* one more product produced and work time spent */
        mmv_inc_value(base, machine[product], working); /* API */
        count[product]->ull += 1;     /* or direct mmap update */

        /* all other products are "queued" for this time */
        for (i = 0; i < ACME_PRODUCTS_COUNT; i++)
            if (i != product)
                mmv_inc_value(base, inqueue[i], working);
    }
```

At this stage, it will be informative to compile and run the complete example program, which can be found in `${PCP_DEMOS_DIR}/mmv/acme.c`. There is an associated `Makefile` to build it, in the same directory. Running the `acme` binary creates the instrumentation shown in Example 4.5, “Memory Mapped Value Reports”, with live values letting us explore simple queueing effects in products being created on the ACME factory floor.

### Example 4.5. Memory Mapped Value Reports

```
              pminfo -m mmv.acme
mmv.acme.products.queuetime PMID: 70.321.10
mmv.acme.products.time PMID: 70.321.8
mmv.acme.products.count PMID: 70.321.7

pmval -f2 -s3 mmv.acme.products.time
metric:    mmv.acme.products.time
host:      localhost
semantics: cumulative counter (converting to rate)
units:     microsec (converting to time utilization)
samples:   3
interval:  1.00 sec

               Anvils               Rockets    Giant_Rubber_Bands
                 0.37                  0.12                  0.50
                 0.35                  0.25                  0.38
                 0.57                  0.20                  0.23
```

Experimentation with the algorithm from Example 4.4, “Memory Mapped Value Updates” is encouraged. In particular, observe the effects of rate conversion (counter metric type) of a metric with units of "time" (PM_TIME_*). The reported values are calculated over a sampling interval, which also has units of "time", forming a utilization. This is extremely valuable performance analysis currency - comparable metrics would include processor utilization, disk spindle utilization, and so forth.

# Elapsed Time Measures

One problem with the instrumentation model embodied by the **pcp_mmv** library is providing timing information for long-running operations. For instrumenting long-running operations, like uploading

downloading a file, the overall operation may be broken into smaller, discrete units of work which can be easily instrumented in terms of operations and througput measures. In other cases, there are no divisible units for long-running operations (for example a black-box library call) and instrumenting these operations presents a challenge. Sometimes the best that can be done is adding the instrumentation point at the completion of the operation, and simply accept the "bursty" nature of this approach. In these problematic cases, the work completed in one sampling-interval may have begun several intervals before, from the point of view of the monitoring tool, which can lead to misleading results.

One technique that is available to combat this is through use of the MMV_TYPE_ELAPSED metric type, which provides the concept of a "timed section" of code. This mechanism stores the start time of an operation along with the mapped metric value (an "elapsed time" counter), via the **mmv_stats_interval_start** instrumentation function. Then, with help from the MMV PMDA which recognizes this type, the act of sampling the metric value causes an **interim** timestamp to be taken (by the MMV PMDA, not the application) and **combined** with the initial timestamp to form a more accurate reflection of time spent within the timed section, which effectively smooths out the bursty nature of the instrumentation.

The completion of each timed section of code is marked by a call to **mmv_stats_interval_end** which signifies to the MMV PMDA that the operation is not active, and no extra "in-progress" time should be applied to the exported value. At that time, the elapsed time for the entire operation is calculated and accounted toward metrics value.

# 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 the former 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.

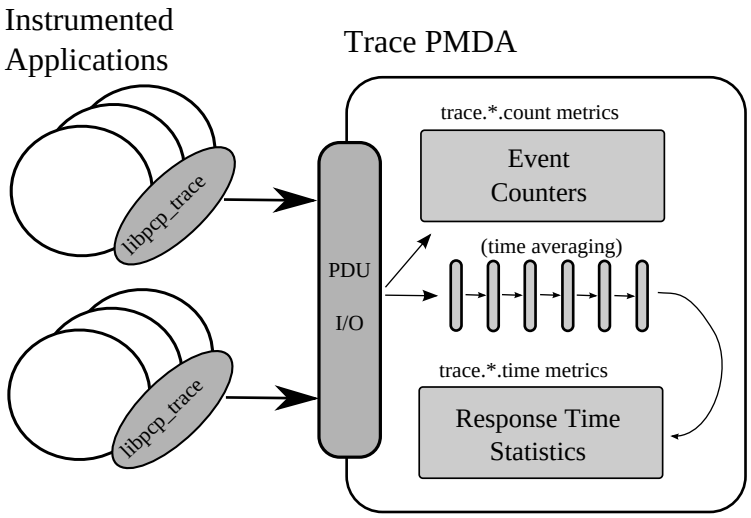
# Trace PMDA Design

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

## Application Interaction

Figure 4.3, “Trace PMDA Overview” describes the general state maintained within the trace PMDA.

**Figure 4.3. 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 the section called “Rolling-Window Periodic Sampling”.

# 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 4.6, “Rolling-Window Sampling Technique”:

**Example 4.6. 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. the section called “Trace API”, documents in more detail all metrics exported by the trace PMDA.

## 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.

## Rolling-Window Periodic Sampling

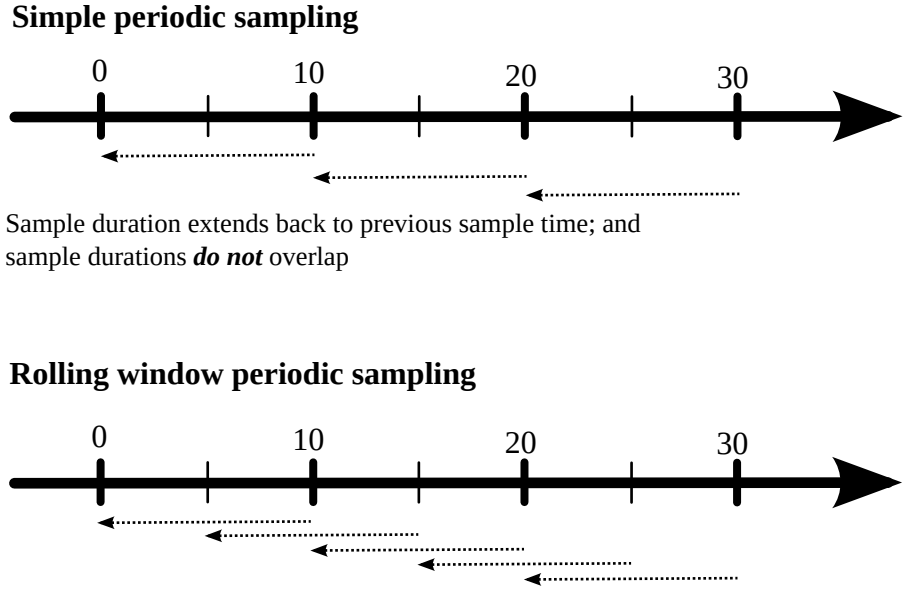
In contrast to simple periodic sampling with its single historical buffer, the rolling-window periodic sampling technique maintains a number of separate buffers. One buffer is marked as the current working buffer, and the remainder of the buffers hold historical data. As each event occurs, the current 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 4.4, “Sample Duration Comparison”.

**Figure 4.4. 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 4.3, “Trace PMDA Overview”).

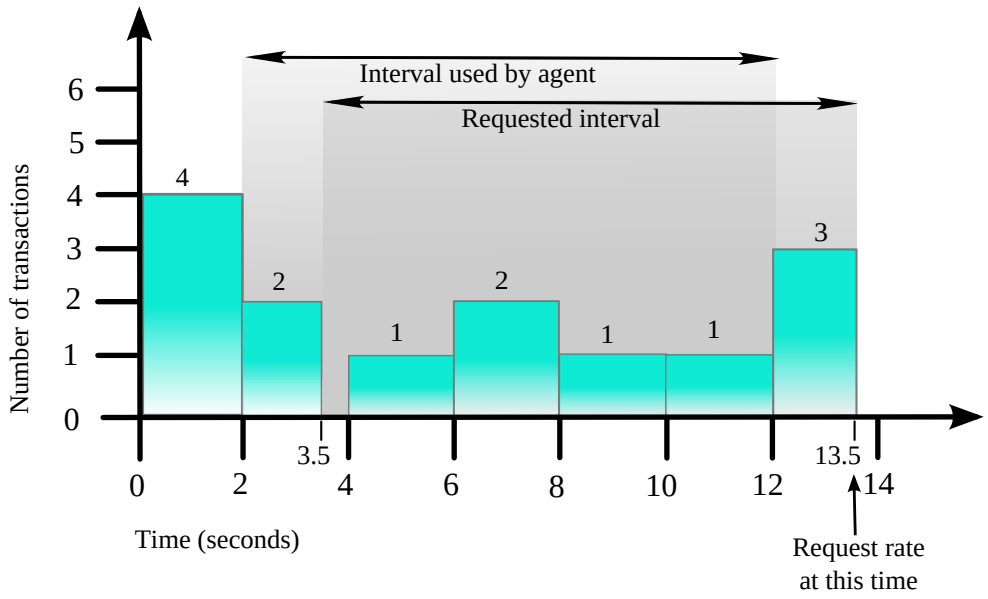
When the current working buffer is moved into the set of historical buffers, the least recent historical buffer is cleared of data and becomes the new working buffer.

## 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 4.5, “Sampling Intervals”.

**Figure 4.5. 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.

# Configuring the Trace PMDA

The trace PMDA is configurable primarily through command-line options. The list of command-line options in Table 4.1, “Selected Command-Line Options” 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. |
| Counter and observation metric units | Since the data being exported by the `trace.observe.value` and `trace.counter.count` 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 **pmchart**. |
| Instance domain refresh | The set of instances exported for each of the `trace` metrics can be cleared through the storable `trace.control.reset` metric. |

# 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.

# 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, " `trace.transact` Metrics":

**Table 4.2. `trace.transact` Metrics**

| Metric | Description |
|---|---|
| `trace.transact.ave_time` | The average service time per transaction type. This time is calculated over the last sample duration. |
| `trace.transact.count` | The running count for each transaction type seen since the trace PMDA started. |

| Metric | Description |
|---|---|
| trace.transact.max_time | The maximum service time per transaction type within the last sample duration. |
| trace.transact.min_time | The minimum service time per transaction type within the last sample duration. |
| trace.transact.rate | The average rate at which each transaction type is completed. The rate is calculated over the last sample duration. |
| trace.transact.total_time | The cumulative time spent processing each transaction since the trace PMDA started running. |

# 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, "`trace.point` Metrics":

### Table 4.3. `trace.point` Metrics

| Metric | Description |
|---|---|
| trace.point.count | Running count of point observations for each tag seen since the trace PMDA started. |
| trace.point.rate | The average rate at which observation points occur for each tag within the last sample duration. |

# 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, "`trace.observe` Metrics":

### Table 4.4. `trace.observe` Metrics

| Metric | Description |
|---|---|
| trace.observe.count | Running count of observations seen since the trace PMDA started. |
| trace.observe.rate | The average rate at which observations for each tag occur. This rate is calculated over the last sample duration. |
| trace.observe.value | The numeric value associated with the observation last seen by the trace PMDA. |
| trace.counter | Counter data is exported through the `trace.counter` metrics. The only difference between `trace.counter` and `trace.observe` metrics is that the numeric value of `trace.counter` must be a monotonic increasing count. |

# Configuring the Trace Library

The trace library is configurable through the use of environment variables listed in Table 4.5, “Environment Variables” as well as through the state flags listed in Table 4.6, “State Flags”. 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, “State Flags”). |
| PCP_TRACE_RECONNECT | A list of values which represents the backoff approach that the `libpcp_trace` 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, “State Flags” 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 |
|---|---|
| PMTRACE_STATE_NONE | 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 `libpcp_trace` routines. |
| PMTRACE_STATE_API | High-level diagnostics. This flag simply displays entry into each of the API routines. |
| PMTRACE_STATE_COMMS | Diagnostic messages related to establishing and maintaining the communication channel between application and PMDA. |
| PMTRACE_STATE_PDU | The low-level details of the trace protocol data units (PDU) is displayed as each PDU is transmitted or received. |
| PMTRACE_STATE_PDUBUF | The full contents of the PDU buffers are dumped as PDUs are transmitted and received. |
| PMTRACE_STATE_NOAGENT | Interprocess communication control. If this flag is set, it causes interprocess communication between the instrumented application and the trace PMDA to be |

| Flag | Description |
| --- | --- |
| | skipped. This flag is a debugging aid for applications using `libpcp_trace`. |
| `PMTRACE_STATE_ASYNC` | 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 `libpcp_trace` entry points. |

# Appendix A. Acronyms

Table A.1, “Performance Co-Pilot Acronyms and Their Meanings” 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 |
| PMDA | Performance Metrics Domain Agent |
| PMID | Performance Metric Identifier |
| PMNS | Performance Metrics Name Space |
| TCP/IP | Transmission Control Protocol/Internet Protocol |

# Index