Performance Co-Pilot™ Programmer's Guide

Document Number 007-3434-002

CONTRIBUTORS

Engineering and written contributions by David Chatterton, Mark Goodwin,
Ken McDonell, Ania Milewska, Nathan Scott, and Tim Shimmin
Edited by Bill Tuthill
Production by Allen Clardy

Performance Co-Pilot™ Programmer’s Guide
Document Number 007-3434-002

Contents

List of Figures

List of Tables

About This Guide

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

Intended Audience

This document describes the programming interfaces to Performance Co-Pilot. It is intended for performance analysts or system administrators who want to extend or customize performance monitoring tools available with PCP, and also for developers who need to integrate their applications into the PCP framework. This book is written for those who are competent with the C programming language, the UNIX operating system, and the target domain from which the desired performance metrics are to be extracted. Familiarity with the Performance Co-Pilot tool suite is assumed—refer to the companion volume *Performance Co-Pilot User's and Administrator's Guide*.

What This Guide Contains

Here is an overview of the material in this book:

- 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 for the Performance Co-Pilot.
- Chapter 3, "PMAPI—The Performance Metrics API," describes the interface that allows you to design custom performance monitoring tools.
- Appendix A, "Acronyms,"contains an explanation of terms and acronyms.

Resources for Further Information

The companion book titled *Performance Co-Pilot User's and Administrator's Guide* describes many of the concepts required to understand PCP. The following reference pages provide useful information for PMDA development: PCPIntro(1), pmcd(1), PMAPI(3), PMDA(3), and pmdatrace(3).

Several examples of PMDA source are provided with PCP, including the trivial PMDA, the simple PMDA, and the example *txmon* PMDA. As their names suggest, they do not provide any useful metrics; rather, they serve as examples of how to implement a PMDA. Complete source code for these PMDAs is located in directories under */usr/pcp/pmdas*, some of which are symbolic links to */var/pcp/pmdas*. Source code for sample applications that use the PCP trace library (*libpcp_trace*) may be found in */var/pcp/demos/trace*.

Several include files are relevant:

- */usr/include/pcp/pmapi.h* and */usr/include/pcp/pmda.h*
- */usr/include/pcp/impl.h* (required only for complex or low-level PMDAs)
- */usr/include/pcp/trace.h*

The Web site http://www.sgi.com/Products/hardware/challenge/CoPilot is worth visiting for updates about the product. A PCP tutorial in HTML format is distributed with the PCP product and may be found in the *pcp.man.tutorial* subsystem.

Conventions Used in This Guide

The table below lists typographic conventions used in this guide.

Purpose	**Example**
Function or subroutine names	Unlike a DSO PMDA, a daemon PMDA has a **main** routine.
Names of shell commands	The *pmcd* daemon requests and collects performance data.
Titles of manuals	See the *Performance Co-Pilot User's and Administrator's Guide*.
Filenames and pathnames	Demo programs are in */var/pcp/demos* and */var/pcp/pmdas*.
What you type (variables in italic)	**`cc -g`** *sourcefile***`.c -lpcp_pmda -lpcp -lgen`**
Exact quotes of computer output	`Error: unknown PMID`
Reference page (man page) name	See pmcd(1)

Chapter 1

Programming Performance Co-Pilot

Introduction

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

Performance Co-Pilot provides unparalleled power to quickly isolate and understand performance behavior, resource utilization, activity levels and performance bottlenecks.

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

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

- By writing a Performance Metrics Domain Agent (PMDA) to collect performance metrics from an uncharted performance domain.
- By creating new analysis or visualization tools using documented routines from the Performance Metrics Application Programming Interface (PMAPI).
- Adding performance instrumentation to an application using the "trace" facilities of the PCP trace library (*libpcp_trace*) and the trace PMDA.

These topics are covered in chapters two, three, and four of this manual.

In addition, the topic of customizing a PCP installation is covered in the "Customizing and Extending the PCP Services" chapter of the companion *Performance Co-Pilot User's and Administrator's Guide*.

Performance Co-Pilot Architecture

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

Performance Co-Pilot consists of several monitoring and collecting tools. Monitoring tools such as *pmchart* and *pmview* visualize metrics but have minimal interaction with target systems; see pmchart(1) and pmview(1). Collection tools (called PMDAs) extract performance values from target systems but do not provide graphical user interfaces.

Systems supporting PCP services are broadly classified into two categories:

- **Collector**: hosts that have the Performance Metrics Collection Daemon (PMCD) and one or more PMDAs running to collect and export performance metrics.
- **Monitor**: hosts that import performance metrics from one or more collector hosts to be consumed by tools to monitor, manage, or record the performance of the collector hosts.

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

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

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

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

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

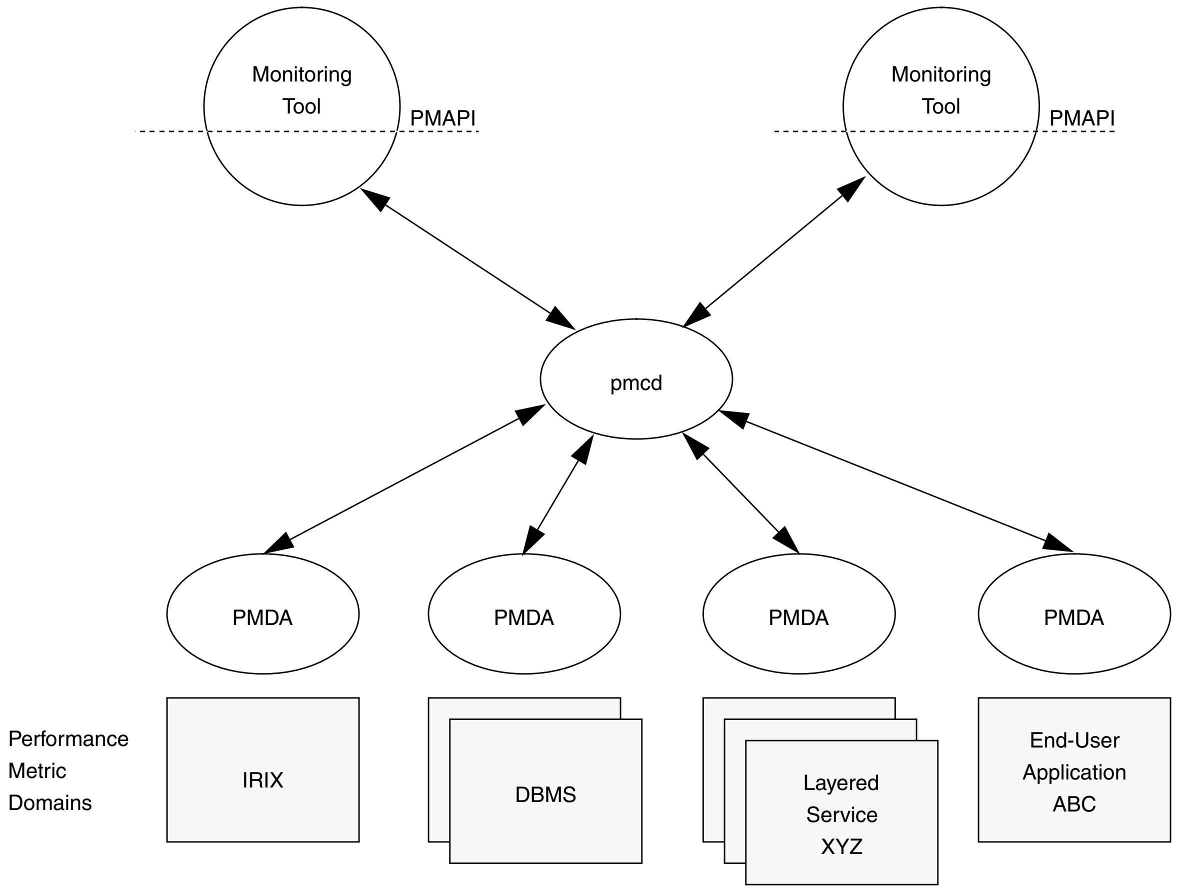

Figure 1-1 PCP Global Process Architecture

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

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

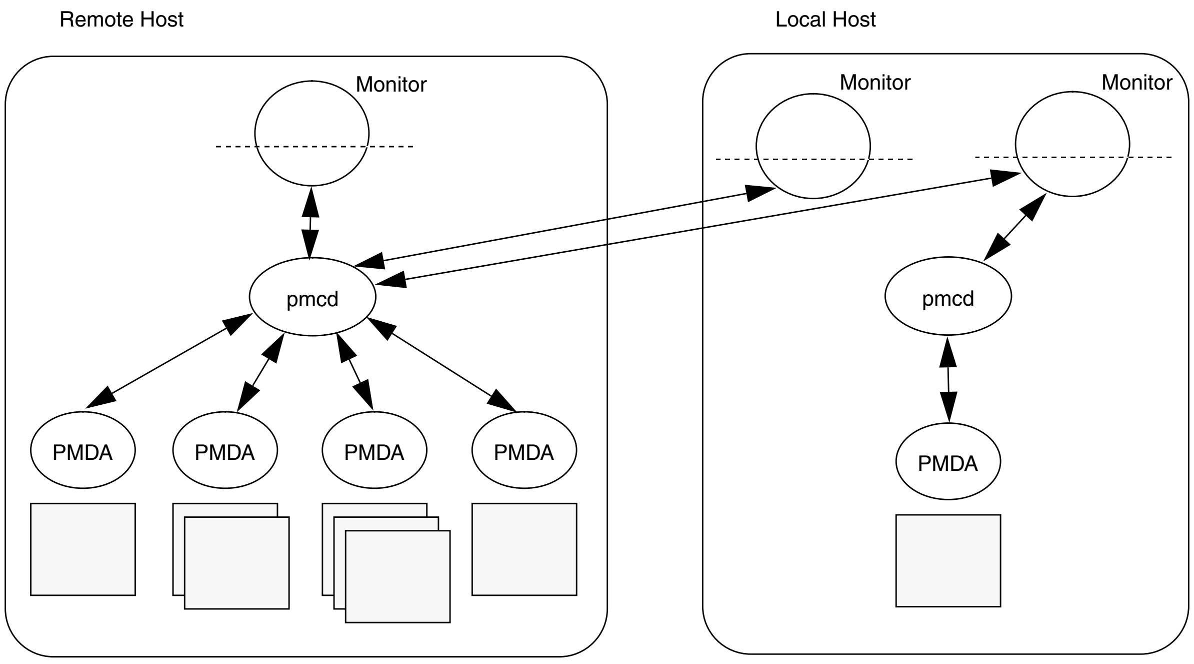

Figure 1-2 Process Structure for Distributed Operation

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 PMAPI(3). Connections between PMDAs and *pmcd* are managed by the PMDA routines; see PMDA(3). There can be multiple monitor clients and multiple PMDAs on the one host, but there may be only one *pmcd* process.

Namespace

Each PMDA provides a domain of metrics, whether they be for IRIX, 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

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

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

Distributed PMNS services necessitated changes to PCP protocols between client applications and *pmcd*, and to the internal format of PCP archive files. Release 2.0 is compatible with earlier releases, so new PCP components operate correctly with either new or old PCP components. For example, when a PCP 2.0 monitoring tool connects to PCP 1.x *pmcd*, or attempts to process a PCP archive created by PCP 1.x *pmlogger*, the monitoring tool reverts to using the local PMNS.

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

Retrospective Sources of Performance Metrics

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

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

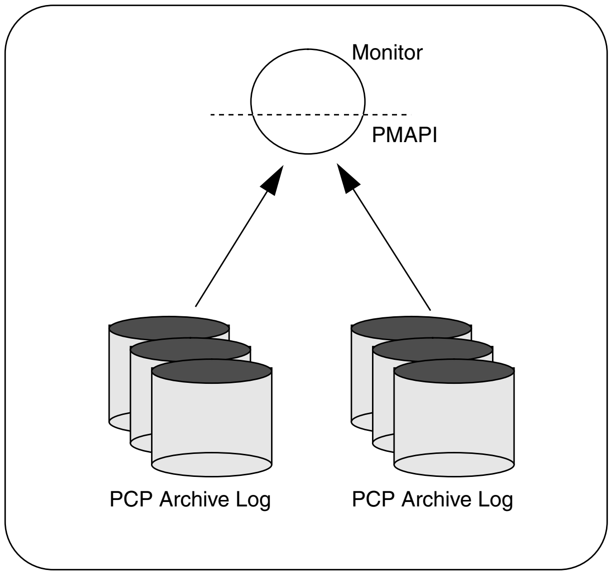

Figure 1-3 Architecture for Retrospective Analysis

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 is a step-by-step guide to writing your own PMDA.

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, so 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 routines.

You can incorporate new performance metrics into the PCP framework by creating a PMDA, then re-configuring *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*.

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

The In-Process (DSO) Method

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

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

The Daemon Process Method

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

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

The Shell Process Method

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

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

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

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 API (PMAPI), documented in Chapter 3. The PMAPI provides performance tool developers with access to all of the distributed services of the Performance Metrics Collection System (PMCS), and is the interface used by the standard PCP utilities.

Source for a sample PMAPI client may be found in the directory */var/pcp/demos/pmclient*.

Chapter 2

Writing a PMDA

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

The presentation assumes the developer is using the standard PCP *libpcp_pmda* library, as documented in the PMDA(3) and associated reference 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(2) file; a sequential log file (where the PMDA parses the tail of the log file to extract the information); a snapshot file (the PMDA re-reads the file as required); or application-specific communication services (IPC). The choice of export methodology is typically determined by the source of the instrumentation (the target domain) rather than by the PMDA.

Procedure Checklist

Here are the suggested steps for designing and implementing a PMDA:

1. Determine how to extract the metrics from the target domain.
2. Select an appropriate architecture for the PMDA (daemon or DSO, IPC, sproc(2)).
3. Define the metrics and instances that the PMDA will support.
4. Implement the functionality to extract the metric values.
5. Assign Performance Metric Identifiers (PMIDs) for the metrics, along with names for the metrics in the Performance Metrics Name Space (PMNS).
6. Specify the help file and control data structures for metrics and instances that are required by the standard PMDA implementation library routines.
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 dbpmda(1).
10. Install and connect the PMDA to a running *pmcd* process; see pmcd(1).
11. Configure or develop tools to use the new metrics. For examples of visualization tools, see pmchart(1), pmgadgets(1) and pmview(1). For examples of alarm tools, see pmie(1) and pmrules(1).

    Where appropriate, define pmlogger(1) configurations suitable for creating PCP archives containing the new metrics.

PMDA Architecture

This section discusses the two methods of connecting a PMDA to a PMCD process: as a separate process using some inter-process communication (IPC) protocol, or as a dynamically attached library (that is, a Dynamic Shared Object or DSO; see the DSO(5) reference page for more details).

Overview

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

For daemon PMDA the following request PDUs must be supported:

- PDU_FETCH—request for metric values; see pmFetch(3).
- PDU_PROFILE—a list of instances required for the corresponding metrics in subsequent fetches; see pmAddProfile(3).
- PDU_INSTANCE_REQ—request for a particular instance domain for instance descriptions; see pmGetInDom(3).
- PDU_DESC_REQ—request for metadata describing metrics; see pmLookupDesc(3).
- PDU_TEXT_REQ—request for metric help text; see pmLookupText(3).
- PDU_RESULT—values to store into metrics; see pmStore(3).

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 routines, a PMDA can be implemented as a dynamically shared object (DSO) and attached by PMCD at run time with the **dlopen** call; see dlopen(3). This eliminates the need for an IPC layer (typically a UNIX **pipe**) between each PMDA and *pmcd*, because each request becomes a function call rather than a message exchange. The required library routines are detailed in the section "DSO Interface" on page 29.

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 routines that are subsequently called with function pointers. When a DSO PMDA is installed, the *pmcd* configuration file */etc/pmcd.conf* is updated to reflect the domain and name of the PMDA, the location of the shared object, and the name of the initialization routine. The initialization sequence is discussed in the section "Initializing a PMDA" on page 37.

Example—Install Simple PMDA as a DSO

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

```
# cd /var/pcp/pmdas/simple
# cat /etc/pmcd.conf
# Name  Id      IPC     IPC Params      File/Cmd
irix    1       dso     irix_init       libirixpmda.so
pmcd    2       dso     pmcd_init       pmda_pmcd.so
proc    3       dso     proc_init       pmda_proc.so
# ./Install
You will need to choose an appropriate configuration for installation
of the "simple" Performance Metrics Domain Agent (PMDA).

collector   collect performance statistics on this system
  monitor     allow this system to monitor local and/or remote systems
  both        collector and monitor configuration for this system

Please enter c(ollector) or m(onitor) or b(oth) [b] both

Updating the Performance Metrics Name Space (PMNS) ...
Installing pmchart view(s) ...
Install simple as a daemon or dso agent? [daemon] dso
...
Check simple metrics have appeared ... 4 metrics and 6 values
# cat /etc/pmcd.conf
# Name  Id      IPC     IPC Params      File/Cmd
irix    1       dso     irix_init       libirixpmda.so
pmcd    2       dso     pmcd_init       pmda_pmcd.so
proc    3       dso     proc_init       pmda_proc.so
simple  253     dso     simple_init     pmda_simple.so
```

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

Daemon PMDA

A DSO PMDA provides the most efficient communication between the PMDA and PMCD. However, this approach has some disadvantages resulting from the DSO PMDA being the same process as *pmcd*, namely:

- An error or bug that causes a DSO PMDA to exit also causes *pmcd* to exit.
- There is only one thread of control in *pmcd*, so a computationally expensive PMDA, or worse, a PMDA that blocks for I/O, adversely affects the performance of *pmcd*.
- As the DSO PMDA is opened with **dlopen**, the PMDA cannot link with any dynamic libraries other than *libc*, *libpcp*, and *libpcp_pmda*.
- The *pmcd* daemon runs as root, so any DSO PMDAs also run as root.

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 routines 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, it is possible to 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**, so a pipe can be easily connected to the PMDA using standard input and output. The *pmcd* process may also connect to a daemon PMDA using TCP/IP or UNIX domain sockets; see inet(7) or unix(7).

Example—Install Simple PMDA as a Daemon

As superuser, install the simple PMDA as a daemon process. As with the previous example, the output may differ due to other PMDAs already installed.

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

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

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

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; see the contents of the */var/pcp/pmdas/cisco* directory and the pmdacisco(1) reference page.

Domains, Metrics, and Instances

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

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

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

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

Tip: Although uniqueness of the domain number in the */etc/pmcd.conf* control file used by pmcd(1) 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 file */var/pcp/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.

The metrics should also 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) reference 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 pmLookupDesc(3):

```
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 fields for

- a unique identifier (Performance Metric Identifier or PMID) that differentiates this metric from other metrics across the union of all PMDAs
- a data type indicator saying whether the format is an integer (32 or 64 bit, signed or unsigned); float; double; string; or arbitrary aggregate of binary data

- an instance domain identifier that links this metric to an instance domain
- an encoding of the value's semantics (counter, instantaneous, or discrete)
- a description of the value's units based on dimension and scale in the three orthogonal dimensions of space, time, and count (or events)

Symbolic constants of the form PM_TYPE_*, PM_SEM_*, PM_SPACE_*, PM_TIME_*, and PM_COUNT_*, defined in */usr/include/pcp/pmapi.h,* may be used to initialize the elements of a *pmDesc*. The type *pmID* 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:

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

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

Note: The three components of the PMID should correspond exactly to the three-part definition of the PMID for the corresponding metric in the PMNS described in "Namespace" on page 25.

A table of *pmdaMetric* structures should be defined within the PMDA, with one structure per metric. This structure contains a *pmDesc* structure and a handle that allows PMDA-specific structures to be associated with each metric:

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

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.

Example—A Single Metric, the Trivial PMDA

The trivial PMDA has only a singular metric (that is, no instance domains):

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

This single metric (*trivial.time*)

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

Semantics

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

- a counter (PM_SEM_COUNTER)
- an instantaneous value (PM_SEM_INSTANT)
- a 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.

Example—The effect of semantics on a metric

For a given time interval covering six consecutive timestamps, each spanning two units of time, the following metric values are exported from a PMDA ("N/A" implies no value is available):

```
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
```

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.

N Dimensional Data

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

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

- mapped onto one dimensional coordinates
- enumerated into the Performance Metrics Name Space (PMNS); for details, see "Naming and Identifying Performance Metrics" on page 50

For example, this 2 x 3 array of values called M can be represented as instances 1,..., 6 for a metric M, or as instances 1, 2, 3 for metric M1 and instances 1, 2, 3 for metric M2.

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

or

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

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

In certain special cases (for example, such as for a histogram), it may be appropriate to export an array of values as raw binary data (the type encoding in the descriptor is PM_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.

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:

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

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

The complete instance domain description is specified in a *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. The *pmInDom* can be safely cast to *_pmInDom_int*, which specifies the PMDA's domain and the instance number within the PMDA:

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

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

An instance domain may also be associated with more than one metric; see pmdaInit(3).

Example—Several Metrics and an Instance Domain, the Simple PMDA

The simple PMDA has four metrics and one instance domain of three instances.

```
/*
 * 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,
        { 0,0,0,0,0,0} }, },
/* color */
    { NULL,
      { PMDA_PMID(0,1), PM_TYPE_32, COLOR_INDOM, PM_SEM_INSTANT,
        { 0,0,0,0,0,0} }, },
/* time.user */
    { NULL,
      { PMDA_PMID(1,2), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        { 0, 1, 0, 0, PM_TIME_SEC, 0 } }, },
/* time.sys */
    { NULL,
      { PMDA_PMID(1,3), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        { 0, 1, 0, 0, PM_TIME_SEC, 0 } }, },
/* now */
    { NULL,
      { PMDA_PMID(2,4), PM_TYPE_U32, NOW_INDOM, PM_SEM_INSTANT,
        { 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 routines and data structures of the *libpcp_pmda* library. Briefly, this involves providing new routines 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 *_pmPMDA* structure during PMDA initialization.

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

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

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: the extraction of the data and communication with PMCD.

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

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

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

Latency and Threads of Control

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

An architecture that has been used successfully for several PMDAs is to create one or more **sproc** child processes to obtain information while the main process communicates with *pmcd*; see sproc(2). At the simplest deployment of this arrangement, the two processes may execute without synchronization.

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

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

One caveat about this style of caching PMDA—it is generally better if the PMDA converts counts to rates based upon consecutive periodic sampling from the underlying instrumentation. By exporting pre-computed 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).

Namespace

The *pmns* file defines the namespace of the PMDA. It is a simple text file that is used during installation to expand the namespace of the PMCD process. The format of this file is described by pmns(4).

Client processes will not be able to access the PMDA's metrics if the *pmns* file is not defined and installed with the *pmnsadd* command; see pmnsadd(1). The installed list of metric names and their corresponding PMIDs can be found in */var/pcp/pmns/root*.

Example—pmns File for the Simple PMDA

The simple PMDA has five metrics: three metrics immediately under the *simple* node, and two metrics under another non-terminal node called *simple.time*:

```
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. For example, an alternative PMNS for the simple PMDA could be as follows:

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

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

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 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 newhelp(1). Converted help text files are usually placed in the PMDA's directory below */var/pcp/pmdas* as part of the PMDA installation procedure.

Example—Help Text for the Simple PMDA

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

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

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

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

```
This counter may be modified with pmstore(1).

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

The value of the metric is monotonic increasing in the range 0 to
255, then back to 0.  The different instances have different starting
values, namely 0 (red), 100 (green) and 200 (blue).

The metric values 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
/var/pcp/pmdas/simple/simple.conf has been modified - if it has then
the file is re-parsed and the instance domain for this metric is again
constructed according to its contents.

This configuration file contains a single line of comma-separated time
tokens from this set:
  “sec”  (seconds after the minute),
  “min”  (minutes after the hour),
  “hour” (hour since midnight).

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

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

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 this example is typically benign:

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

- If a metric lacks support in a particular version of the underlying instrumentation, the PMDA should respond to **pmFetch** requests with a *pmResult* in which no values are returned for the unsupported metric. This is marginally friendlier than the other semantically acceptable option of returning an "illegal PMID" error, or PM_ERR_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 -EACCES if a user or application tries to amend the value of an unsupported metric.
- The value extraction, conversion, and printing routines (**pmExtractValue**, **pmConvScale**, **pmAtomStr**, **pmTypeStr**, and **pmPrintValue**) will return the error PM_ERR_CONV, or an appropriate diagnostic string, if an attempt is made to operate on a value for which the *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 the *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 routines.

DSO Interface

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

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. In order 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.

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

- pmdaFetch(3)
- pmdaProfile(3)
- pmdaInstance(3)
- pmdaDesc(3)
- pmdaText(3)
- pmdaStore(3)

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 routines in *libpcp.so* and *libpcp_pmda.so*.

Example—trivial_fetchCallBack in the Trivial PMDA

The trivial PMDA uses all of the default callbacks. The additional callback for **pmdaFetch** is defined as **trivial_fetchCallBack**:

```
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;
    else if (inst != PM_IN_NULL)
        return PM_ERR_INST;

    atom->l = time(NULL);
    return 0;
}
```

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

Example—simple_fetchCallBack in the Simple PMDA

The simple PMDA callback for **pmdaFetch** is more complicated because it must support more metrics, some metrics are instantiated with each fetch, and one instance domain is dynamic. The default **pmdaFetch** callback 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**:

```
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. Metric *simple.numfetch* has no instance domain and is easily handled first:

```
static int
simple_fetchCallBack(pmdaMetric *mdesc, unsigned int inst, pmAtomValue *atom)
{
    int                 i;
    static int          oldfetch = 0;
    static struct tms   tms;
    __pmID_int          *idp = (__pmID_int *)&(mdesc->m_desc.pmid);

    if (inst != PM_IN_NULL &&
        !(idp->cluster == 0 && idp->item == 1) &&
        !(idp->cluster == 2 && idp->item == 4))
        return PM_ERR_INST;

    if (idp->cluster == 0) {
        if (idp->item == 0) {                   /* simple.numfetch */
            atom->l = numfetch;
        }
```

For the metric *simple.color* the *inst* parameter is used to specify which instance is required:

```
        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;
```

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

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

Finally the *simple.now* metric is in a third cluster and uses *inst* again to select a specific instance from the INDOM_NOW instance domain:

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

Example—simple_store in the Simple PMDA

The simple PMDA permits some of the metrics it supports to be modified by **pmStore**; see pmstore(1). The **pmdaStore** callback (which returns -EACCESS to indicate no metrics can be altered) is replaced by **simple_store** in **simple_init**. This replacement function must take the same arguments so that it can be assigned to the function pointer in the *pmdaInterface* structure.

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

```
static int
simple_store(pmResult *result, pmdaExt *pmda)
{
    int         i, j, val, sts = 0;
    pmAtomValue av;
    pmValueSet  *vsp = NULL;
    __pmID_int  *pmidp = NULL;

    for (i = 0; i < result->numpmid; i++) {
        vsp = result->vset[i];
        pmidp = (__pmID_int *)&vsp->pmid;

        if (pmidp->cluster == 0) {  /* storable metrics are cluster0 */
            switch (pmidp->item) {
                case 0:                            /* simple.numfetch */
                    val = vsp->vlist[0].value.lval;
                    if (val < 0) {
                        sts = PM_ERR_SIGN;
                        val = 0;
                    }
                    numfetch = val;
                    break;

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

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

```
                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 cluster 0 that are not supported by the simple PMDA should result in an error value of PM_ERR_PMID:

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

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

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

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

PMDA Structures

PMDA structures used with the *pcp_pmda* library are defined in */usr/include/pcp/pmda.h*.

pmdaInterface

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

```
typedef struct {
    int domain;     /* set/return performance metrics domain id here */
    struct {
        unsigned int pmda_interface : 8;  /* PMDA DSO version */
        unsigned int pmapi_version : 8;   /* PMAPI version */
        unsigned int flags : 16;          /* usage TBD */
    } comm;             /* set/return communication and version info */
    int status;         /* return initialization status here */

    union {

/* Interface Version 1 (PCP 1.0 & PCP 1.1) */

        struct {
            int     (*profile)(__pmProfile *);
            int     (*fetch)(int, pmID *, pmResult **);
            int     (*desc)(pmID, pmDesc *);
            int     (*instance)(pmInDom, int, char *, __pmInResult **);
            int     (*text)(int, int, char **);
            int     (*control)(pmResult *, int, int, int);
            int     (*store)(pmResult *);
        } one;

/* Interface Version 2 (PCP 2.0) */

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

    } version;
```

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

pmdaExt

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

```
typedef struct {
    unsigned int e_flags;       /* usage TBD */
    void        *e_ext;         /* usage TBD */

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

    pmdaResultCallBack e_resultCallBack; /* to clean up pmResult after fetch */
    pmdaFetchCallBack  e_fetchCallBack;  /* to assign metric values in fetch */
    pmdaCheckCallBack  e_checkCallBack;  /* callback on receipt of a PDU */
    pmdaDoneCallBack   e_doneCallBack;   /* callback after PDU is processed */
} pmdaExt;
```

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

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 "DSO PMDA" on page 11, an initialization function is provided by a DSO PMDA and called by PMCD. Using the standard PMDA wrappers, the same routine can also be used as part of the daemon PMDA initialization. This PMDA initialization function is responsible for

- 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** routine; see pmdaInit(3).

Because the initialization routine 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** routine; see pmdaDSO(3).

Example—trivial_init in the Trivial PMDA

The trivial PMDA has no instances and a single callback for the **pmdaFetch** routine called **trivial_fetchCallBack**; see pmdaFetch(3):

```
void trivial_init(pmdaInterface *dp)
{
    pmdaSetFetchCallBack(dp, trivial_fetchCallBack);

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

The trivial PMDA is always installed as a daemon PMDA.

Example—simple_init 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.

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**:

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

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

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

Daemon Initialization

In addition to the initialization routine that can be shared by a DSO and a daemon PMDA, a daemon PMDA must also

- 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 routines in the *pcp_pmda* library; see pmdaDaemon(3), pmdaGetOpt(3), pmdaOpenLog(3), pmdaConnect(3), and pmdaMain(3).

Example—main in the Simple PMDA

The simple PMDA requires no additional command-line arguments other than those handled by **pmdaGetOpt**; see pmdaGetOpt(3):

```
int
main(int argc, char **argv)
{
    int                 err = 0;
    int                 c = 0;
    pmdaInterface       dispatch;
    char                *p;

    /* trim cmd name of leading directory components */
    pmProgname = argv[0];
    for (p = pmProgname; *p; p++) {
        if (*p == '/')
            pmProgname = p+1;
    }

    isDSO = 0;

    pmdaDaemon(&dispatch, PMDA_INTERFACE_2, pmProgname, SIMPLE,
               "simple.log", "/var/pcp/pmdas/simple/help");
    if ((c = pmdaGetOpt(argc, argv, "D:d:i:l:pu:?", &dispatch, &err)) != EOF)
        err++;

    if (err)
        usage();
    pmdaOpenLog(&dispatch);
    simple_init(&dispatch);
    simple_timenow_check();
    pmdaConnect(&dispatch);
    pmdaMain(&dispatch);

    exit(0);
    /*NOTREACHED*/
}
```

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. If the PMDA dumps core, *dbx* and related tools (see dbx(1)) cannot reasonably explore the generated core image, which includes the *pmcd* image and any other active DSO PMDAs.

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

If you suspect a PMDA has been terminated due to a time out failure, check the *pmcd* log file, usually */var/adm/pcplog/pmcd.log*.

A more robust way of testing a *pmcd* 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 dbpmda(3) for details.

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

Debugging Information

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

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

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

Most of the *pcp_pmda* library routines log additional information if the `DBG_TRACE_LIBPMDA` flag is set within the PMDA; see PMDA(3). The command-line argument **-D** is trapped by **pmdaGetOpt** to set the global debugging control variable *pmDebug*. Adding tests within the PMDA for the trace flags `DBG_TRACE_APPL0`, `DBG_TRACE_APPL1`, and `DBG_TRACE_APPL2` 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 standard error. By convention, all debugging information is enclosed by preprocessor `#ifdef DEBUG` statements so that they can be compiled out of the program at a later stage, if required.

Example—Log Stores Into simple.numfetch in the Simple PMDA

By adding this segment of code to **simple_store**, whenever *pmstore* (see pmstore(1)) attempts to change *simple.numfetch* and *pmDebug* has the `DBG_TRACE_APPL0` flag set, a log message is sent to the current log file:

```
case 0: /* simple.numfetch */
    val = vsp->vlist[0].value.lval;
    if (val < 0) {
        sts = PM_ERR_SIGN;
        val = 0;
    }
```

```
#ifdef DEBUG
        if (pmDebug & DBG_TRACE_APPL0) {
            fprintf(stderr,
                 "simple: %d stored into numfetch", val);
        }
#endif
        numfetch = val;
        break;
```

dbpmda Debug Utility

The *dbpmda* utility provides a simple interface to the PDU communication protocol. It allows daemon and DSO PMDAs to be tested with most request types, while the PMDA process may be monitored with *dbx*, *par* and other diagnostic tools. The reference page dbpmda(1) contains a sample session with the simple PMDA.

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 pmdatrace(3).

The monitoring of transactions using 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** function has similar semantics, but allows an arbitrary numeric value to be passed to the trace PMDA.

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

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

Integration of PMDA

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

The PMDA typically would have its own directory below */var/pcp/pmdas* into which several files would be installed. In the description in "Installing a PMDA" on page 43, the PMDA of interest is assumed to be known by the name **newbie**, hence the PMDA directory would be */var/pcp/pmdas/newbie*.

Note: Any installation or removal of a PMDA involves updating files and directories that are typically well protected. Hence the procedures described in this section must be executed as 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 */var/pcp/pmdas/newbie*.
- The namespace has been updated in the directory */var/pcp/pmns*.
- The PMDA binary has been installed, usually in the directory */var/pcp/lib* for a DSO PMDA, or in the PMDA directory */var/pcp/pmdas/newbie* for a daemon PMDA.
- The */etc/pmcd.conf* file has been updated.
- The *pmcd* process has been restarted or notified (with a SIGHUP signal) that the new PMDA exists.

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

The *Makefile* should include an **install** target to compile and link the PMDA (as a DSO, or a daemon or both) in the PMDA directory, and in the case of a DSO PMDA, install the shared library in */var/pcp/lib*. The **clobber** target should remove any files created as a by-product of the **install** target.

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

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

The *Install* script should make use of the generic procedures defined in the script */usr/pcp/lib/pmdaproc.sh*, and may be as straightforward as the one used for the trivial PMDA, namely:

```
# Get the common procedures and variable assignments
#
. /usr/pcp/lib/pmdaproc.sh

# The name of the PMDA
#
iam=trivial

# Do it
#
_setup
_install

exit 0
```

The following variables may be assigned values to modify the behavior of the `_setup` and `_install` procedures from */usr/pcp/lib/pmdaproc.sh*.

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

Variable	**Use**	**Default**
iam	Name of the PMDA; assignment to this variable is mandatory. 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`
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.	

Table 2-1 (continued) Variables to Control Behavior of Generic pmdaproc.sh Procedures

Variable	Use	Default
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$iam.so`
dso_entry	The name of the initialization function for a DSO PMDA.	`$iam_init`
domain	The numerical PMDA domain number (from *domain.h*).	
SYMDOM	The symbolic name of the PMDA domain number (from *domain.h*).	

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

The variables may also have their assignments changed by the user’s response to the common prompt:

```
You will need to choose an appropriate configuration for installation
of the ... Performance Metrics Domain Agent (PMDA).

  collector   collect performance statistics on this system
  monitor     allow this system to monitor local and/or remote systems
  both        collector and monitor configuration for this system
```

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

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

- Using default assignments, and interaction where ambiguity exists, determine the PMDA type (DSO or daemon) and the IPC parameters, if any.
- Copy the `$pmns_source` file, replacing symbolic references to `SYMDOM` by the desired numeric domain number from `domain`.
- Merge the PMDA's namespace into the PCP namespace at the non-leaf node identified by `$pmns_name`.
- If any *pmchart* views can be found (files with names ending in ".pmchart"), copy these to the standard directory (*/var/pcp/config/pmchart*) with the ".pmchart" suffix removed.
- Create new help files from `$help_source` after replacing symbolic references to `SYMDOM` by the desired numeric domain number from `domain`.
- Terminate the old daemon PMDA, if any.
- Use the *Makefile* to build the appropriate executables.
- Add the PMDA specification to *pmcd*'s configuration file (*/etc/pmcd.conf*).
- Notify *pmcd*. To minimize the impact on the services *pmcd* provides, sending a SIGHUP to *pmcd* forces it to reread the configuration file and start, restart, or remove any PMDAs that have changed since the file was last read.
- Check that the metrics from the new PMDA are available.

There are some PMDA changes that may trick PMCD into thinking nothing has changed, and not restarting the PMDA. Most notable are changes to the PMDA executable. In these cases, you may need to explicitly remove the PMDA (see below), or more drastically, restart *pmcd* as follows:

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

Example—PMDA Install Scripts

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

Upgrading a PMNS to Include Metrics From a New PMDA

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

Removing a PMDA

The simplest way to stop a PMDA from running, apart from killing the process, is to remove the entry from */etc/pmcd.conf* and signal PMCD to re-read 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 above.

Example—PMDA Remove Scripts

The files */var/pcp/pmdas/*/Remove* 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 providing “canned” configuration files that monitor key metrics of the new PMDA, users can quickly see the performance of the target system, as characterized by key metrics in the new PMDA.

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

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

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

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.

Templates for parameterized alarm configurations can be specified using the *pmrules* command; see pmrules(1), and pmie(1). Rules involving metrics from the new PMDA may be created directly.

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

Chapter 3

PMAPI—The Performance Metrics API

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

The PMAPI is a set of functions and data structure definitions that allow client applications to access performance data from one or more Performance Metric 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 Performance Co-Pilot.

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. Performance Co-Pilot 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 `irix.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 “Domains, Metrics, and Instances” on page 15.

For brevity and efficiency, internally PCP avoids using ASCII names for performance metrics, and instead uses an identification scheme that unambiguously associates a single integer with each known performance metric. This integer is known as a Performance Metric Identifier, or PMID. For routines 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 “Metrics” on page 17.

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

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

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

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

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.

For example, consider the following;

```
$ pminfo -f irix.filesys.free

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

The metric `irix.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/root*, */dev/usr* and */dev/dsk/dks0d2s0*. These instances form an instance domain that is shared by the performance metrics `irix.filesys.capacity`, `irix.filesys.used`, `irix.filesys.free`, `irix.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 `irix.mem.freemem` always has exactly one value. Note that this is semantically different to a performance metric like `irix.kernel.percpu.syscall` that has a non-singular instance domain, but may have only one value available; for example, on a system with a single processor.

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

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

Note: Some instance domains are more dynamic than others. For example, consider the instance domains behind the performance metrics `proc.memory.physical.dat` (one instance per process), `irix.swap.free` (one instance per swap partition) and `irix.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 an archive log of performance data collected by *pmlogger* at some remote host or earlier time (a retrospective or archive source). The metric's source is specified when the PMAPI context is created by calling the **pmNewContext** function.

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

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

Performance Metric Descriptions

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

```
/* 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 */
```

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

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

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

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

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

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

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

This information is encoded in the *pmUnits* data structure, which is embedded in the *pmDesc* structure:

```
/*
 * Encoding for the units (dimensions and
 * scale) for Performance Metric Values
 *
 * For example, a pmUnits struct of
 * { 1, -1, 0, PM_SPACE_MBYTE, PM_TIME_SEC, 0 }
 * represents Mbytes/sec, while
 * { 0, 1, -1, 0, PM_TIME_HOUR, 6 }
 * represents hours/million-events
 */
typedef struct {
   int dimSpace:4;   /* space dimension */
   int dimTime:4;    /* time dimension */
   int dimCount:4;   /* event dimension */
   int scaleSpace:4; /* one of PM_SPACE_* below */
   int scaleTime:4;  /* one of PM_TIME_* below */
   int scaleCount:4; /* one of PM_COUNT_* below */
} pmUnits;   /* dimensional units and scale of value */
/* pmUnits.scaleSpace */
#define PM_SPACE_BYTE 0  /* bytes */
#define PM_SPACE_KBYTE 1 /* Kilobytes (1024) */
#define PM_SPACE_MBYTE 2 /* Megabytes (1024^2) */
#define PM_SPACE_GBYTE 3 /* Gigabytes (1024^3) */
#define PM_SPACE_TBYTE 4 /* Terabytes (1024^4) */

/* pmUnits.scaleTime */
#define PM_TIME_NSEC 0   /* nanoseconds */
#define PM_TIME_USEC 1   /* microseconds */
#define PM_TIME_MSEC 2   /* milliseconds */
#define PM_TIME_SEC 3    /* seconds */
#define PM_TIME_MIN 4    /* minutes */
#define PM_TIME_HOUR 5   /* hours */

/*
 * pmUnits.scaleCount (e.g. count events, syscalls,
 * interrupts, etc.) -- these are simply powers of 10,
 * and not enumerated here.
 * e.g. 6 for 10^6, or -3 for 10^-3
 */
#define PM_COUNT_ONE 0 /* 1 */
```

Performance Metrics Values

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

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

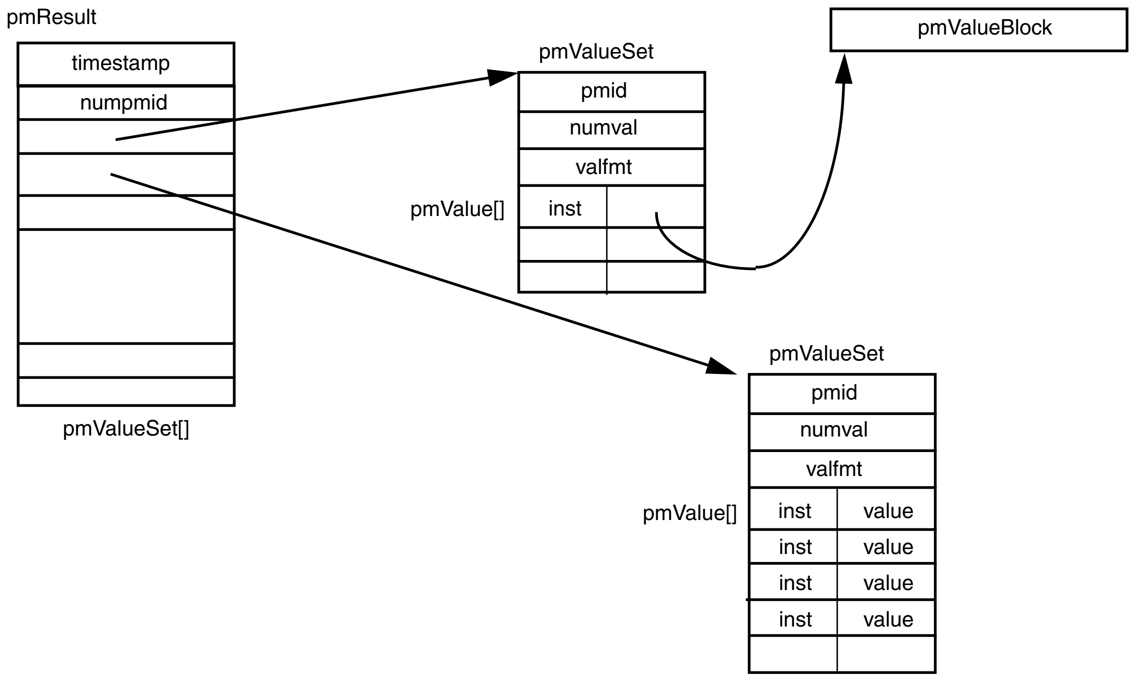

Figure 3-1 A Structured Result for Performance Metrics From pmFetch

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* located via *pval*:

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

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

```
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;
```

A *pmValueSet* contains all of the values to be returned from *pmFetch* for a single performance metric identified by the *pmid* field. 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 pmErrStr(3)) 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*.

```
/* 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
```

The *pmResult* structure contains a timestamp and an array of *numpmid* pointers to *pmValueSets*. 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 timestamp with each *pmResult* that serves to identify when the performance metric values were collected. The time is in the format returned by **gettimeofday** and is typically very close to the time when the metrics are exported across the PMAPI.

Note: There is a question of exactly "when" individual metrics may have been collected, especially given their origin in potentially different performance metric domains, and variability in metric updating frequency by individual PMDAs. PCP uses a pragmatic approach, in which the PMAPI implementation returns all metrics with values accurate as of the timestamp, 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 "PMAPI Ancillary Support Services" on page 77.

General Issues of PMAPI Programming Style and Interaction

The following sections specify the programming style used in the PMAPI:

- "Variable Length Argument and Results Lists"
- "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** and **pmFetch**), the underlying implementation uses dynamic allocation through **malloc** in the called routine, 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 routine (**pmFreeResult**) to release the storage, due to the complexity of the data structure and the need to make multiple calls to **free** in the correct sequence. As a general rule, if the called routine returns an error status, then no allocation is done, the pointer to the variable sized result is undefined, and **free** or **pmFreeResult** should not be called.

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 routine along the lines of *strerror* is provided to translate error conditions into error messages; see *pmErrStr*. The error condition is returned as the function value from a previous PMAPI call; there is no global error indicator (unlike *errno*). This is an attempt to anticipate and accommodate a programming environment that does not hinder the implementation of multi-threaded performance tools.

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

```
pmerr -l
```

PMAPI Procedural Interface

The following sections describe all of the PMAPI routines 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 Metrics Services"
- "PMAPI Archive-Specific Services"

PMAPI Name Space Services

pmGetChildren

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

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

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

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

pmGetChildrenStatus

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

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

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

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

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

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

pmGetPMNSLocation

```
int pmGetPMNSLocation(void)
```

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

pmLoadNameSpace

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

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

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

Note: The distributed PMNS services in PCP 2.0 avoid the need for a local PMNS in most cases, so applications typically would *not* use **pmLoadNameSpace**. If applications do not call **pmLoadNameSpace**, the default PMNS is the one at the source of the performance metrics.

pmLookupName

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

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

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

pmNameID

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

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

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

pmTraversePMNS

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

The routine **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 routine **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 strdup(3C).

pmTrimNameSpace

```
int pmTrimNameSpace(void)
```

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

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

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

The PMNS services in PCP 2.0 avoid the need for a local PMNS in most cases (and by default use only the PMNS of the metrics in a PCP archive) so applications would typically *not* call **pmTrimNameSpace**.

pmUnloadNameSpace

```
int pmUnloadNameSpace(void)
```

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

PMAPI Metric Description Services

pmLookupDesc

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

Given a Performance Metrics Identifier as *pmid*, **pmLookupDesc** returns the associated *pmDesc* structure through the parameter *desc* from the current PMAPI context. For more information about *pmDesc*, see "Performance Metric Descriptions" on page 53.

pmLookupText

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

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

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 malloc(3C) and free(3C).

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 "PMDA Help Text" on page 26.

pmLookupInDomText

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

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

The 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 dialog. 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 malloc(3C) and free(3C).

The help text files used to implement **pmLookupInDomText** are created using *newhelp* and accessed by the appropriate PMDA response to requests forwarded to the PMDA by *pmcd*. Further details may be found in "PMDA Help Text" on page 26.

PMAPI Instance Domain Services

pmGetInDom

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

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

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

pmLookupInDom

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

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

pmNameInDom

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

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

PMAPI Context Services

The following table 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		
pmDelProfile	yes	yes		
pmDupContext	yes	yes	yes	
pmFetch	yes	yes	yes	
pmFetchArchive	yes		yes	(1)
pmGetArchiveEnd	yes			(1)
pmGetArchiveLabel	yes			(1)
pmGetChildren	yes			(5)
pmGetChildrenStatus	yes			(5)
pmGetPMNSLocation	yes			
pmGetInDom	yes		yes	(2)

Table 3-1 (continued) Context Components of PMAPI Functions

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

Notes:

1. Operation supported only for PMAPI contexts where the source of metrics is an archive.
2. A specific instance domain is included in the arguments to these routines, and the result is independent of the instance profile for any PMAPI context.
3. The metadata that describes a performance metric is sensitive to the source of the metrics, but independent of any instance profile and of the collection time.
4. Operation supported only for PMAPI contexts where the source of metrics is a host. The text associated with a metric is assumed to be invariant with time and is definitely insensitive to the current members of the instance domain. In all cases this information is unavailable from an archive context (it is not included in the archive logs), and is directly available from a PMDA via *pmcd* in the other cases.
5. PMNS service routines using a local PMNS do *not* depend on the PMAPI context, whereas PCP 2.0 distributed PMNS services are dependent on the source of metrics.

6. This operation is supported only for contexts where the source of the metrics is a host. Further, the instance identifiers are included in the argument to the routine, 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 routine.

pmNewContext

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

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

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

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

Once established, the association between a PMAPI context and a source of metrics is fixed for the life of the context; however, routines 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**.

pmDestroyContext

```
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.

pmDupContext

```
int pmDupContext(void)
```

Replicate the current PMAPI context (source, instance profile, and collection time). This routine 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

```
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

```
int pmWhichContext(void)
```

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

pmAddProfile

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

Add new instance specifications to 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 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).
```

pmDelProfile

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

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

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

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

pmSetMode

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

This routine 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 routines: **pmFetch**, **pmFetchArchive**, **pmLookupDesc**, **pmGetInDom**, **pmLookupInDom** and **pmNameInDom**.

The **pmSetMode** routine 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 timestamp. 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 timestamp returned through the *pmResult*. The **pmSetMode** parameter *delta* defines an additional number of milliseconds 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 fixed sampling rate.

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

For example, the following code fragment may be used to dump only those values stored in correct temporal sequence, for the specified performance metric *my.metric.name*:

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

```
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);
    }
```

pmReconnectContext

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

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

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

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

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

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.

PMAPI Metrics Services

pmFetch

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

The most common PMAPI operation is likely to be calls to **pmFetch**, specifying a list of PMIDs (for example, as constructed by **pmLookupName**) through *pmidlist* and *numpmid*. The call to **pmFetch** is executed in the context of a source of metrics, instance profile, and collection time, previously established by calls to the routines described in "PMAPI Context Services" on page 65.

The principal result from **pmFetch** is returned as a tree structured *result*, described in the section "Performance Metrics Values" on page 56.

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 pmerr(1) and pmErrStr(3). 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.

The following 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:

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

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

Note: If a response is not received back from PMCD within 10 seconds, the **pmFetch** will time out and return 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 environment variable PMCD_REQUEST_TIMEOUT; see PCPIntro(1).

pmFreeResult

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

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

pmStore

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

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

The routine **pmStore** 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 Archive-Specific Services

pmGetArchiveLabel

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

Provided the current PMAPI context is associated with a PCP archive log, the **pmGetArchiveLabel** function may be used to fetch the label record from the archive. The structure returned through *lp* is as follows:

```
/*
 * Label Record at the start of every log file - as exported above the PMAPI ...
 */
#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[40];         /* $TZ at collection host */
} pmLogLabel;
```

pmGetArchiveEnd

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

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

pmGetInDomArchive

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

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

This routine is a specialized version of the more general PMAPI routine **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 malloc(3C) and free(3C).

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).

pmLookupInDomArchive

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

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

This routine is a specialized version of the more general PMAPI routine *pmLookupInDom*.

The **pmLookupInDomArchive** routine returns a positive instance identifier on success.

pmNameInDomArchive

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

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

This routine is a specialized version of the more general PMAPI routine **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 malloc(3C) and free(3C).

pmFetchArchive

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

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

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

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 pmtime(1).

This service is most useful when processing PCP archive logs, to control parameters such as the current archive position, update interval, replay rate, and timezone, but it can also be used in live mode to control a subset of these parameters.

Applications such as *pmchart*, *pmview*, *oview*, and *pmval* use the time control services to connect to an instance of the time control server process, *pmtime*, which provides a uniform graphical user interface to the time control services.

A full description of the PMAPI time control functions along with code examples can be found in the reference page pmtime(3).

PMAPI Ancillary Support Services

The routines 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.

pmErrStr

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

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

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

pmExtractValue

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

The *pmValue* structure is embedded within the *pmResult* structure, which is used to return one or more performance metrics; see the description of **pmFetch**.

All performance metric values may be encoded in a *pmAtomValue* union, defined as follows:

```
/* 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 routine **pmExtractValue** 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 **pmFetch**). 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 3-2 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 that some of the conversions involving the types PM_TYPE_STRING and PM_TYPE_AGGREGATE are indeed possible, but are marked N; the rationale is that **pmExtractValue** should not attempt to duplicate functionality already available in the C library through **sscanf** and **sprintf**.

Table 3-2 PMAPI Type Conversion

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

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

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

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.

pmConvScale

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

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

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

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

pmUnitsStr

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

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

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

For example, if **pu* were *{1, -2, 0, PM_SPACE_MBYTE, PM_TIME_MSEC, 0}*, then the result string would be "Mbyte/sec^2."

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

pmIDStr

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

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

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

pmInDomStr

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

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

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

pmTypeStr

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

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

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

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

pmAtomStr

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

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

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

pmPrintValue

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

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 *type*.

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.

The following example illustrates using **pmPrintValue** to print the values from a *pmResult* structure returned via **pmFetch**:

```
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));
```

pmSortInstances

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

The routine **pmSortInstances** 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.

PMAPI Programming Issues and Examples

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

The source code for a sample client (*pmclient*) using the PMAPI is shipped as part of the *pcp.sw.demo* subsystem of the Performance Co-Pilot product. See the pmclient(1) reference page, and the source code, located in */var/pcp/demos/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 macro *BINGO* with the name "irix.sys.statistic.bingo", and then be able to use *BINGO* to get at the value of the corresponding metric.

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

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

```
# one line comment
mystuff {
    irix.sys.statistic.bingo  BINGO
    oracle.latchstats.lru.miss  MISSED
}
```

The above *pmgenmap*(1) input produces the following C code, suitable for including with the `#include` statement:

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

Initializing New Metrics

Using the code generated by *pmgenmap*, we are now able to easily initialize the application's metric specifications as follows:

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

pmLookupName(numpmid, mystuff, pmidlist);
```

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

Iterative Processing of Values

Assuming the tool is required to report values every five seconds, use code similar to the following:

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

Accommodating Program Evolution

The flexibility provided by the PMAPI and the *pmgenmap* utility is demonstrated by this example. Consider the requirement for reporting a third metric "irix.sys.boring.new" (an instantaneous value) in the middle of the two already reported. Add this line to the middle of the specification file:

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

Then regenerate the *#include* file, and amend the **printf** statement as follows:

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

Handling PMAPI Errors

The following simple but complete PMAPI application demonstrates the recommended style for handling PMAPI error conditions:

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

int
main(int argc, char* argv[])
{
    int                 sts = 0;
    char                *host = "localhost";
    char                *metric = "irix.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;
    }
```

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

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

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

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

Acronyms

This chapter provides a glossary of the acronyms used in the Performance Co-Pilot documentation, help cards, reference pages, and user interface.

Table A-1 Performance Co-Pilot Acronyms and Their Meanings

Acronym	Meaning
DBMS	Database Management System
DSO	Dynamic Shared Object
IP	Internet Protocol
I/O	Input/Output
IPC	Inter-process 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

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3434-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389