

PMDF Programmer's Reference Manual

Order Number: N-5303-66-NN-N

February 2020

This document describes the PMDF Application Program Interface (API) and callable SEND facility for version 6.8 of the PMDF software.

Revision/Update Information: This manual supersedes the *V6.7 PMDF Programmer's Reference Manual*

Software Version: PMDF V6.8

Operating System and Version: Red Hat Enterprise Linux 7 or later on x86_64; (or other compatible Linux distribution)

OpenVMS Alpha V7.3-2 or later;

OpenVMS I64 V8.2 or later;

Copyright ©2020 Process Software, LLC.
Unpublished — all rights reserved under
the copyright laws of the United States

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means electronic, mechanical, magnetic, optical, chemical, or otherwise without the prior written permission of:

Process Software, LLC
959 Concord Street
Framingham, MA 01701-4682 USA
Voice: +1 508 879 6994; FAX: +1 508 879 0042
info@process.com

Process Software, LLC ("Process") makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Furthermore, Process Software reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Process Software to notify any person of such revision or changes.

Use of PMDF, PMDF-MSGSTORE, PMDF-MTA, and/or PMDF-TLS and associated documentation is authorized only by a Software License Agreement. Such license agreements specify the number of systems on which the software is authorized for use, and, among other things, specifically prohibit use or duplication of software or documentation, in whole or in part, except as authorized by the Software License Agreement.

Restricted Rights Legend

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or as set forth in the Commercial Computer Software — Restricted Rights clause at FAR 52.227-19.

The PMDF mark and all PMDF-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries and are used under license.

ALL-IN-1, Alpha AXP, AXP, Bookreader, DEC, DECnet, HP, I64, IA64, Integrity, MAILbus, MailWorks, Message Router, MicroVAX, OpenVMS, Pathworks, PSI, RMS, TeamLinks, TOPS-20, Tru64, TruCluster, ULTRIX, VAX, VAX Notes, VMScluster, VMS, and WPS-PLUS are registered trademarks of Hewlett-Packard Company.

AS/400, CICS, IBM, Office Vision, OS/2, PROFS, and VTAM are registered trademarks of International Business Machines Corporation. CMS, DISOSS, OfficeVision/VM, OfficeVision/400, OV/VM, and TSO are trademarks of International Business Machines Corporation.

dexNET is a registered trademark of Fujitsu Imaging Systems of America, Inc.

FaxBox is a registered trademark of DCE Communications Group Limited.

InterConnections is a trademark of InterConnections, Inc.

LANmanager and Microsoft are registered trademarks of Microsoft Corporation.

MHS, Netware, and Novell are registered trademarks of Novell, Inc.

PGP and Pretty Good Privacy are registered trademarks of Pretty Good Privacy, Inc.

Attachmate is a registered trademark and PathWay is a trademark of Attachmate Corporation.

PostScript is a registered trademark of Adobe Systems Incorporated.

SPARC is a trademark of SPARC International, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Gold-Mail is a trademark of Data Processing Design, Inc.

libedit/editline is Copyright (c) 1992, 1993, The Regents of the University of California. All rights reserved.

AlphaMate is a registered trademark of Motorola, Inc.

cc:Mail is a trademark of cc:Mail, Inc., a wholly-owned subsidiary of Lotus Development Corporation. Lotus Notes is a registered trademark of Lotus Development Corporation.

RC2 and RC4 are registered trademarks of RSA Data Security, Inc.

Ethernet is a registered trademark of Xerox Corporation.

GIF and "Graphics Interchange Format" are trademarks of CompuServe, Incorporated.

InterDrive is a registered trademark of FTP Software, Inc.

Memo is a trade mark of Verimaton ApS.

LaserJet and PCL are registered trademarks of Hewlett-Packard Company.

Jnet is a registered trademark of Wingra, Inc.

Pine and Pico are trademarks of the University of Washington, used by permission.

Solaris, Sun, and SunOS are trademarks of Sun Microsystems, Inc.

TCPware and MultiNet are registered trademarks of Process Software.

TIFF is a trademark of Aldus Corporation.

Copyright (c) 1990-2000 Sleepycat Software. All rights reserved.

Contents

PREFACE

ix

CHAPTER 1	THE PMDF API	1-1
1.1	INTRODUCTION TO THE API ROUTINES	1-1
1.2	ENQUEUING MESSAGES	1-2
1.3	DEQUEUING MESSAGES	1-4
1.4	MULTIPLE MESSAGE ENQUEUE AND DEQUEUE CONTEXTS	1-7
1.5	USAGE FROM MULTI-THREADED PROCESSES	1-7
1.6	MESSAGE HEADER STRUCTURES	1-7
1.7	PROGRAMS THAT RUN INDEFINITELY	1-10
1.7.1	OpenVMS Considerations _____	1-11
1.7.2	UNIX Considerations _____	1-11
1.8	WRITING OUTPUT FROM A CHANNEL PROGRAM	1-12
1.9	DEBUGGING PROGRAMS AND LOGGING MESSAGING ACTIVITY	1-12
1.10	REQUIRED PRIVILEGES	1-12
1.10.1	OpenVMS Systems _____	1-12
1.10.2	UNIX Systems _____	1-13
1.11	COMPILING AND LINKING PROGRAMS	1-13
1.12	EXAMPLES OF USING THE API ROUTINES	1-13
1.12.1	Enqueuing a Simple Message _____	1-14
1.12.2	Dequeuing Messages _____	1-16
1.12.3	Dequeuing & Re-enqueuing Messages _____	1-23
1.12.4	Dequeuing & Returning Messages _____	1-30
1.13	API ROUTINE DESCRIPTIONS	1-36
1.13.1	Summary of Routines _____	1-36
1.13.2	Order Dependencies _____	1-38
1.13.3	Strings Passed To and From the API _____	1-40
1.13.4	Routine Descriptions _____	1-41
	PMDFABORTMESSAGE	1-42
	PMDFABORTPROGRAM	1-43
	PMDFADDHEADERLINE	1-45
	PMDFADDRECIPIENT	1-47
	PMDFADDRESSDISPOSE	1-50
	PMDFADDRESSGET	1-51
	PMDFADDRESSGETPROPERTY	1-53
	PMDFADDRESSPARSELIST	1-55
	PMDFALIASNOEXPANSION	1-57
	PMDFCANCELCALLBACK	1-58
	PMDFCLOSELOGFILE	1-59
	PMDFCLOSEQUEUECACHE	1-60
	PMDFCOPYMESSAGE	1-61
	PMDFDATABASEADDEENTRY	1-63
	PMDFDATABASECLOSE	1-67
	PMDFDATABASEDELETEENTRY	1-68
	PMDFDATABASEGETENTRY	1-70

Contents

PMDFDEBUG	1-73
PMDFDECODEMESSAGE	1-75
PMDFDEFERMESSAGE	1-78
PMDFDELETEHEADERLINE	1-80
PMDFDEQUEUEEND	1-82
PMDFDEQUEUEINITIALIZE	1-83
PMDFDEQUEUEMESSAGE	1-84
PMDFDEQUEUEMESSAGEEND	1-85
PMDFDISPOSECHANNELCOUNTERS	1-87
PMDFDISPOSEHEADER	1-88
PMDFDONE	1-89
PMDFENQUEUEINITIALIZE	1-90
PMDFENQUEUEMESSAGE	1-91
PMDFGETADDRESSPROPERTY	1-93
PMDFGETBLOCKSIZE	1-96
PMDFGETCHANNELCOUNTERS	1-97
PMDFGETCHANNELNAME	1-102
PMDFGETDATETIME	1-105
PMDFGETENVELOPEID	1-107
PMDFGETERRORTEXT	1-109
PMDFGETHOSTNAME	1-111
PMDFGETMESSAGE	1-113
PMDFGETMESSAGEID	1-115
PMDFGETPOSTMASTERADDRESS	1-117
PMDFGETRECIPIENT	1-119
PMDFGETRECIPIENTFLAGS	1-122
PMDFGETUNIQUESTRING	1-124
PMDFGETUSERNAME	1-126
PMDFINITIALIZE	1-128
PMDFLOG	1-130
PMDFMAPPINGAPPLY	1-132
PMDFMAPPINGLOAD	1-135
PMDFOPTIONDISPOSE	1-137
PMDFOPTIONGETINTEGER	1-138
PMDFOPTIONGETREAL	1-140
PMDFOPTIONGETSTRING	1-142
PMDFOPTIONREAD	1-144
PMDFQUEUECACHEEND	1-146
PMDFQUEUECACHEGETENTRY	1-147
PMDFREADFFAILURELOG	1-151
PMDFREADHEADER	1-153
PMDFREADLINE	1-154
PMDFREADTEXT	1-156
PMDFRECEIPTCONTROL	1-158
PMDFRECIPIENTDISPOSITION	1-161
PMDFRETURNMESSAGE	1-164
PMDFREWINDMESSAGE	1-167
PMDFSETCALLBACK	1-168
PMDFSETENVELOPEID	1-170
PMDFSETLIMITS	1-172
PMDFSETMUTEX	1-174
PMDFSETRECIPIENTFLAGS	1-177
PMDFSETRECIPIENTTYPE	1-179
PMDFSETRECEIPTADDRESSES	1-181
PMDFSTARTMESSAGEBODY	1-183
PMDFSTARTMESSAGEENVELOPE	1-184
PMDFSTARTMESSAGEHEADER	1-186

PMDFWRITE DATE	1-187
PMDFWRITE FROM	1-188
PMDFWRITE HEADER	1-190
PMDFWRITE LINE	1-191
PMDFWRITE SUBJECT	1-193
PMDFWRITE TEXT	1-195

CHAPTER 2	CALLABLE SEND	2-1
2.1	SENDING A MESSAGE	2-1
2.1.1	Envelope & Header "From:" Address _____	2-2
2.1.2	To:, Cc:, and Bcc: Addresses _____	2-2
2.1.3	Message Headers & Content _____	2-3
2.2	WRITING OUTPUT FROM A CHANNEL PROGRAM	2-4
2.3	REQUIRED PRIVILEGES	2-4
2.4	COMPILING AND LINKING PROGRAMS	2-5
2.5	EXAMPLES OF USING CALLABLE SEND	2-5
2.5.1	Sending a Simple Message _____	2-5
2.5.2	Specifying an Initial Message Header _____	2-7
2.5.3	Multiple Recipients, FAX Addresses, and Per Address Status Messages _____	2-10
2.5.4	Using an Input Procedure _____	2-15
2.6	SUMMARY OF PMDF_send ITEM CODES	2-17
2.7	PMDF_send ROUTINE DESCRIPTION	2-21
	PMDF_send	2-22

APPENDIX A	ERROR CODES	A-1
-------------------	--------------------	------------

GLOSSARY	Glossary-1
-----------------	-------------------

INDEX

EXAMPLES

1-1	Sample Mail Message File _____	1-3
1-2	Enqueuing a Message (Pascal) _____	1-14
1-3	Enqueuing a Message (C) _____	1-15
1-4	Output of Examples 1-2 and 1-3 _____	1-16
1-5	Message Dequeuing (Pascal) _____	1-17
1-6	Message Dequeuing (C) _____	1-20
1-7	Output of Examples 1-5 and 1-6 _____	1-24
1-8	Message Dequeuing & Re-enqueuing (Pascal) _____	1-25
1-9	Message Dequeuing & Re-enqueuing (C) _____	1-27
1-10	Dequeuing & Returning Messages (Pascal) _____	1-31
1-11	Dequeuing & Returning Messages (C) _____	1-32
1-12	Output of Examples 1-10 and 1-11 _____	1-34
2-1	Sending a Simple Message (Pascal) _____	2-5

Contents

2-2	Sending a Simple Message (C) _____	2-6
2-3	Output of Examples 2-1 and 2-2 _____	2-7
2-4	Specifying an Initial Message Header (Pascal) _____	2-8
2-5	Specifying an Initial Message Header (C) _____	2-8
2-6	Input File Used in Examples 2-4 and 2-5 _____	2-9
2-7	Output of Examples 2-4 and 2-5 _____	2-10
2-8	Multiple Addresses (Pascal) _____	2-11
2-9	Multiple Addresses (C) _____	2-13
2-10	Address Status Messages Produced by Examples 2-8 and 2-9 _____	2-14
2-11	Using an Input Procedure (Pascal) _____	2-15
2-12	Using an Input Procedure (C) _____	2-16

FIGURES

1-1	Sample Header Structure _____	1-9
1-2	Calling Precedence for the API Message Enqueue Routines _____	1-39
1-3	Calling Precedence for the API Message Dequeue Routines _____	1-40

TABLES

1-1	Routines Included in the PMDF API _____	1-36
1-2	String Size Constants Used by the API _____	1-41
1-3	Properties of the Address <i>phrase</i> <@otherhost:user@host> _____	1-54
1-4	Database Symbolic Names and Values _____	1-64
1-5	Channel Counters List Entry _____	1-99
1-6	Envelope To: Address NOTARY Flags _____	1-123
1-7	PMDF_queue_cache_get_entry Item Codes _____	1-149
1-8	Disposition Values for Use with PMDF_recipient_disposition _____	1-162
2-1	PMDF_send Item Code Summary _____	2-18

Preface

Purpose of This Manual

This manual describes the PMDF Application Program Interface (API) and callable SEND facility. While this document is primarily intended for system programmers writing mail software, system managers wanting to become more familiar with the inner workings of PMDF may also benefit from a casual reading of this manual. Readers are assumed to be familiar with PMDF and the electronic messaging standards it employs.¹

This manual does not provide a description of PMDF suitable for end users. Non-privileged users cannot use the routines described in this manual as most PMDF operations require sufficient privileges to access messages in the PMDF message queues as well as to create PMDF processing jobs.

Overview of This Manual

This manual describes two distinct interfaces. The first, called simply “the PMDF API”, is a low-level interface which can be used to both enqueue and dequeue PMDF messages. The second interface, referred to as “callable SEND”, is a single, high-level routine which can be used to submit (*i.e.*, enqueue) messages to the PMDF mail system.

Programmers writing code to merely send mail will probably find callable SEND sufficient for their needs. Programmers wanting to write gateways or channels should use the PMDF API. Both interfaces may be used simultaneously.

This manual consists of two chapters:

Chapter 1, *The PMDF API*, describes the low-level interface routines used for enqueueing and dequeuing messages to and from PMDF’s message queues.

Chapter 2, *Callable Send* describes the high-level interface routine used to send (enqueue) mail messages.

Printed copies of this manual can be obtained from Process Software, LLC:

Process Software, LLC
959 Concord Street
Framingham, MA 01701 USA
+1 508 879 6994
+1 508 879 0042 (FAX)
sales@process.com

¹ Generally speaking, RFCs 822, 1123, and 2045–2049.

1 The PMDF API

The PMDF Application Program Interface (API) is composed of the routines described in this chapter. The API can be used to submit to or remove messages from PMDF's message queues. The act of submitting a message to a message queue is called *enqueueing* while removing a message from a queue is called *dequeueing*. User interfaces¹ enqueue messages in order to send mail; while programs that interface with other networks and mail systems dequeue messages to remove them from the queues. Some intermediate processing programs, such as document converters, can both dequeue and enqueue messages.

Note: The callable SEND interface can be used simultaneously with the PMDF API routines.

1.1 Introduction to the API Routines

Each routine in the PMDF API has two calling formats: a Pascal-style format and a C-style format. The only difference between the two is the mechanism used to pass string data: the Pascal-style format uses string descriptors, the C-style format uses pointers to strings. All routines return VMS-style status codes - if the low bit is set, the routine was successful. The strings returned by the C-style routines are null-terminated, but strings passed in to those routines need not be.

The API routines fall into three classes: routines to enqueue a message, routines to dequeue a message, and miscellaneous routines which typically query or set PMDF states. The use of the enqueue and dequeue routines is discussed at length in Sections 1.2 and 1.3.

A working knowledge of RFC 822² and the relevant sections of RFC 1123³ is essential to programmers writing software which will create electronic mail messages with PMDF. Programmers interested in creating MIME-compliant messages should also familiarize themselves with RFCs 2045 and 2046.⁴

Note that channel programs written using the API should always use the `PMDFlog` routine to write output to the channel log file.

¹ User interfaces that send mail are generally referred to as User Agents (UAs).

² A copy of RFC 822, *Standard for the Format of ARPA Internet Text Messages* can be found in the RFC subdirectory of the PMDF documentation directory, `PMDF_ROOT:[DOC.RFC]` on OpenVMS or `/pmdf/doc/rfc` on UNIX and NT.

³ A copy of RFC 1123, *Internet Host Requirements — Application and Support* can be found in the RFC subdirectory of the PMDF documentation directory.

⁴ A copy of these RFCs can be found in the RFC subdirectory of the PMDF documentation directory.

1.2 Enqueuing Messages

Messages are introduced to the PMDF mail system by enqueuing them. Each enqueued message contains two required pieces and one optional piece: the message envelope, the message header, and the optional message body. The contents of the first two pieces, envelope and header, must be provided by the program using the API. The third piece, the message body, is optional - a message does not need to contain a body. Briefly, these three pieces are as follows:

- *Envelope:* The message envelope contains the envelope `From:` address and the list of envelope `To:` addresses. The envelope is created by PMDF when the message is enqueued; the addresses to be placed in the envelope must conform to RFC 822. Note that in the message envelope no distinction is made between `To:`, `Cc:`, and `Bcc:` addresses. Consequently, the envelope `To:` addresses are often referred to as simply envelope recipient addresses.

Programs should treat the message envelope as an opaque structure and rely solely upon the PMDF API routines to read and write information from and to the envelope. The format of the envelope is subject to change; the API routines insulate programmers from such changes.

The routines `PMDFstartMessageEnvelope`, `PMDFsetRecipientType`, and `PMDFaddRecipient` are used to specify the message envelope.

- *Header:* The message header contains RFC 822-style header lines. The program enqueuing the message has nearly complete control over the contents of the header and can specify as many or as few header lines as it sees fit. The only header lines which a program using the API must explicitly generate are the `From:` and `Date:` header lines. If the `From:` header line is omitted, PMDF will construct it from the envelope `From:` address. Note that this may not always be appropriate.⁵ If the `Date:` header line is omitted, PMDF will supply it as well as a `Date-warning:` header line. These two header lines can be generated with `PMDFwriteFrom` and `PMDFwriteDate`.

When the message is enqueued, PMDF will do its best to supply any mandatory header lines that are missing. PMDF will also take measures to ensure that the contents of the header lines conform to any relevant standards.

Any addresses appearing in the message header should conform to RFC 822.

The header is typically written line-by-line with the `PMDFwriteLine` or `PMDFwriteText` routines. It may also be built up and output with the header structure manipulation routines described in Section 1.6. The routines `PMDFwriteFrom`, `PMDFwriteDate`, and `PMDFwriteSubject` can be used to write `From:`, `Date:`, and `Subject:` header lines. Using information supplied via the routines `PMDFstartMessageEnvelope` and `PMDFaddRecipient`, PMDF will generate the `From:` and `To:` header lines automatically as well as any necessary `Cc:` and `Bcc:` header lines.

- *Body:* The optional message body contains the content of the message. As with the message header, the program enqueuing the message has nearly complete control over the contents of the message body. The only exception to this is when the message is structured with multiple parts or requires encoding (e.g., contains binary data, lines

⁵ For instance, when mail is addressed to a mailing list which specifies an `Errors-to:` address, then the `Errors-to:` address should be used as the envelope `From:` address. In this case, it is not appropriate to derive the header `From:` line from the envelope `From:` address.

requiring wrapping, *etc.*). In such cases, PMDF will ensure that the message body conforms to the MIME standard (RFC 2045–2049).

Message body lines are written with `PMDFwriteLine` or `PMDFwriteText` and read with `PMDFreadLine` or `PMDFreadText`.

Enqueued messages are ASCII text files located in the PMDF queue directories.⁶ A sample message is shown in Example 1–1. The essential pieces in that example are: the message envelope, ❶; the message header, ❷; and the message body, ❸.

Example 1–1 Sample Mail Message File

```
m;GONZALO@EXAMPLE.COM ❶
ALONSO@EXAMPLE.COM

Date: Sat, 4 May 2012 18:04 EDT ❷
From: Gonzalo <GONZALO@EXAMPLE.COM>
To: King Alonso <ALONSO@EXAMPLE.COM>
Subject: Walking

Alonso, ❸
  By'r lakin, I can go no further, sir;
  My old bones ache: here's a maze trod indeed
  Through forth-rights and meanders! By your patience,
  I needs must rest me.

                                Gonzalo
```

Note: Do not attempt to directly access messages in the PMDF message queues. Always use the API routines (or callable `SEND`) to access PMDF messages. The file structure of messages in PMDF's message queues is subject to change. In addition, site specific constraints can be placed on messages in various queue directories (*e.g.*, message size, encoding, character set usage, *etc.*). The API routines automatically handle constraints and other issues.

The steps required to enqueue one or more messages are as follows:

1. Initialize PMDF resources and data structures with `PMDFinitialize`.
2. Initialize the PMDF enqueuing subsystem with `PMDFenqueueInitialize`.
3. For each message to enqueue, perform the following steps:
 - a. specify the message envelope with `PMDFstartMessageEnvelope` and `PMDFaddRecipient`;
 - b. specify the message header with `PMDFstartMessageHeader`, `PMDFwriteFrom`, `PMDFwriteDate`, `PMDFwriteSubject`, and `PMDFwriteLine`;
 - c. specify the message body with `PMDFstartMessageBody` and `PMDFwriteLine`; and
 - d. submit the message with `PMDFenqueueMessage`.

⁶ Actually, PMDF-FAX and PMDF-X.400 messages are binary files.

The PMDF API

Enqueuing Messages

4. Deallocate PMDF resources and data structures with `PMDFdone`.

If no message body is to be supplied, then Step 3c can be omitted.

Prior to the `PMDFenqueueMessage` call, a message submission can be aborted at any point in Step 3 by calling either `PMDFabortMessage` or `PMDFdone`. `PMDFabortMessage` only aborts the specified message enqueue while allowing other messages to be enqueued. `PMDFdone` both aborts all active message enqueues and deallocates PMDF resources, which prevents any further enqueue attempts until PMDF is initialized again.

When calling `PMDFstartMessageEnvelope`, a channel name may be specified. The message is then enqueued under the context of the specified channel (*i.e.*, submitted as though enqueued by that channel itself). Typically, the 1 (local) channel should be used. If you are writing your own channel, then you should specify the name of your channel as reported by `PMDFgetChannelName`.⁷

If the message being enqueued is the result of dequeuing a message, then the envelope identification can be copied over from the old message to the new with `PMDFgetEnvelopeId` and `PMDFsetEnvelopeId`. Similarly, the NOTARY processing flags should be copied with `PMDFgetRecipientFlags` and `PMDFsetRecipientFlags`.

Examples 1–2, 1–3, 1–8 and 1–9 all illustrate how to enqueue a message.

Note: On OpenVMS the special `PMDF_*` logicals used to specify the contents of specific header lines and signature boxes are only supported for use with VMS MAIL and the PMDF SEND utility. These logicals are ignored when messages are enqueued by mechanisms other than VMS MAIL.

1.3 Dequeuing Messages

Messages stored in PMDF's message queues are removed from those queues by dequeuing them. This is typically done by channel programs.⁸ When a message is dequeued, it is literally removed from PMDF's message queues and, as far as PMDF is concerned, no longer exists. This means that dequeuing a message relieves PMDF of all further responsibility for the message—the responsibility is assumed to have been passed on to another mailer, gateway, or user agent.

⁷ In some cases, it can be necessary to hard-code a channel name into a program or obtain the channel name by a means other than `PMDFgetChannelName`. For example, the channel name for TCP/IP slave channels is specified at compile time, and PhoneNet slave channels prompt for the name of the channel they are to process.

⁸ Channel programs comprise a broad class of programs that interface PMDF to other networks, mail systems (MTAs), and user agents (UAs). Gateways are an example of channel programs: channel programs which gateway or otherwise transport mail out of PMDF do so by dequeuing messages and are sometimes referred to as *master channels*; channel programs which gateway or otherwise transport mail into PMDF do so by enqueueing messages and are sometimes referred to as *slave channels*. Channel programs can also perform intermediate roles by dequeuing messages from one message queue and requeueing them to another while processing the message at the same time (*e.g.*, converting the message body from one format to another).

The PMDF API Dequeuing Messages

The message queue serviced by a program is determined from “out-of-band” information. For instance, under OpenVMS the queue to be serviced is determined through the `PMDF_CHANNEL` logical whose translation value gives the name of the channel to service. On UNIX and NT, the channel name is given by the `PMDF_CHANNEL` environment variable.

The steps taken to dequeue messages are as follows:

1. Initialize PMDF resources and data structures with `PMDFinitialize`.
2. Initialize the PMDF dequeuing subsystem with `PMDFdequeueInitialize`.
3. Process all pending messages for the channel by repeatedly executing the following steps:
 - a. Access a message with `PMDFgetMessage`. This step also reads the envelope `From:` address from the accessed message.
 - b. Process the accessed message. The following steps are used to read the currently accessed message:
 - i. the envelope `To:` addresses and processing flags are read by repeatedly calling `PMDFgetRecipient` and `PMDFgetRecipientFlags`;
 - ii. the message header lines are read by repeatedly calling `PMDFreadLine` or `PMDFreadText` until the first blank line is encountered, or by calling `PMDFreadHeader` to read the entire header at once; and
 - iii. the message body is read by repeatedly calling `PMDFreadLine` or `PMDFreadText`.
 - iv. any message delivery failure log can be read with repeated calls to `PMDFreadFailureLog`.
 - c. Set the disposition of each envelope `To:` address with repeated calls to `PMDFrecipientDisposition`.
 - d. Dequeue the message with `PMDFdequeueMessageEnd`.
4. Close the message dequeuing subsystem with a call to `PMDFdequeueEnd`.
5. Deallocate PMDF resources and data structures with a call to `PMDFdone`.

Note that the message is not actually dequeued until the very last processing step, 3d. This is very important: it keeps mail from being lost if the channel program fails unexpectedly, the system crashes, or other unexpected disasters occur. The message processing involved in Step 3 can be almost anything. The processing can even involve re-enqueuing the message to another channel as illustrated in Examples 1–8 and 1–9.

When the disposition of each envelope `To:` address is determined, it should be reported to PMDF by calling `PMDFrecipientDisposition`. The recognized dispositions are given in the description of the `PMDFrecipientDisposition` routine and are repeated below.

The PMDF API

Dequeuing Messages

Symbolic name	Value	Description
PMDF_DISP_DEFERRED	1	Recipient address processing has failed because of a temporary problem (e.g., network down, remote host unreachable, mailbox busy, etc.); defer processing of this address until later.
PMDF_DISP_DELIVERED	2	Recipient address was successfully delivered; generate a delivery status notification if required.
PMDF_DISP_FAILED	3	Recipient address processing has failed because of a permanent problem (e.g., invalid recipient address, recipient over quota, etc.); no further delivery attempts should be made for this address. Generate a non-delivery notification if required.
PMDF_DISP_RELAYED	4	Recipient address was forwarded to another address or gatewayed into a non-NOTARY mail system. The message's NOTARY information was preserved - there is no need to generate a "relayed" notification message.
PMDF_DISP_RELAYED_FOREIGN	5	Recipient address was forwarded to another address or gatewayed to a non-NOTARY mail system. The message's NOTARY information was not preserved - generate a "relayed" notification message if required.
PMDF_DISP_RETURN	6	For this recipient, return the message as undeliverable; generate a non-delivery notification if required.

When `PMDFdequeueMessageEnd` is called, the resulting processing depends upon the disposition of the envelope `To:` recipient addresses as reported with `PMDFrecipientDisposition`. If all recipient addresses have a permanent disposition (`PMDF_DISP_DELIVERED`, `PMDF_DISP_FAILED`, `PMDF_DISP_RELAYED`, `PMDF_DISP_RELAYED_FOREIGN`, or `PMDF_DISP_RETURN`), then any required notifications are generated and the message is permanently removed from the processing queue. If all recipients are to be deferred `PMDF_DISP_DEFERRED`, then no notifications are generated and the message is left in the queue for later re-processing. If some recipients have a permanent disposition while others were deferred, then

1. Notifications are generated for those recipients with permanent dispositions and requiring notifications,
2. A new message is enqueued for just those recipients who were deferred, and
3. The original message is removed from the processing queue.

If the program needs to abort message processing, it should call `PMDFdequeueMessageEnd` with a value of `true` (1) for the **defer** argument to that routine. This will leave the message in the processing queue for later re-processing.

In the loop represented by Step 3, `PMDFgetMessage` will repeatedly return each message in the current queue that requires processing until there are no more messages to be processed. Each message in the queue will only be presented once; *i.e.*, a job will not repeatedly see a message that it has deferred. Multiple jobs can simultaneously run and process the same message queue: PMDF will automatically prevent two or more jobs from simultaneously processing the same message. When `PMDFgetMessage` is called, the accessed message is locked so that no other jobs can access that message. The message is unlocked when `PMDFdequeueMessageEnd` is called, or when the job exits (abnormally or otherwise).

Generally, programs which perform dequeue processing do not run indefinitely but rather exit after processing all messages in a specific queue. If it's necessary to write a program that never exits and does dequeue processing, then `PMDFdequeueEnd`, `PMDFcloseQueueCache`, `PMDFcloseLogFile` should be called after looping over every message in a message queue. When it's time to try processing the message queue again, `PMDFdequeueInitialize` should be called before the first `PMDFgetMessage` call.

Examples 1-5, 1-6, 1-8, 1-9, 1-10, and 1-11 all illustrate message dequeue processing.

1.4 Multiple Message Enqueue and Dequeue Contexts

All of the message enqueue and dequeue routines make use of context variables. Each context variable is used to keep track of a single “thread” of message enqueue or dequeue operations. By using multiple context variables, a program can manage and perform several simultaneous message enqueue and dequeue operations. While each enqueue context controls only a single message submission, each dequeue context can control an entire series of message dequeues (*e.g.*, with a single dequeue context all message for a given channel can be processed and dequeued).

1.5 Usage from Multi-threaded Processes

With the exception of the `PMDFdatabase` routines, the PMDF API and underlying routines are re-entrant and thread-safe. Multithreaded routines that will be using the PMDF API must call `PMDFsetMutex` before calling any other API routines, including `PMDFinitialize`. The `PMDFsetMutex` routine provides PMDF with routines to create, lock, unlock, and dispose of thread mutexes. See the description of `PMDFsetMutex` for further details.

For each PMDF database, a single per-process read context is maintained by PMDF. Because of this, any sequence of chained `PMDFdatabaseGetEntry` calls must not be interrupted by other threads accessing the same database. Any interruption will disrupt the read state. A chained sequence is one that starts with a `PMDF_DATABASE_GET_FIRST` or `PMDF_DATABASE_GET_FIRST_ROOT` access followed by `PMDF_DATABASE_GET_NEXT` or `PMDF_DATABASE_GET_NEXT_ROOT` access to find subsequent, related entries.

Note that access to the PMDF queue cache database is thread safe.

1.6 Message Header Structures

A message header structure is used to store a collection of header lines. The stored header lines can be output by `PMDFwriteHeader` to one or more messages being enqueued, and altered with `PMDFaddHeaderLine` or `PMDFdeleteHeaderLine`.

The PMDF API

Message Header Structures

A header structure can be created in one of two ways:

1. While dequeuing a message, the header lines of that message can be read into a header structure with `PMDFreadHeader`. In this case, `PMDFreadHeader` creates a header structure, reads header lines into it, and then returns a pointer to the structure. The structure can then be used with any of the other header routines.
2. By calling `PMDFaddHeaderLine` to add a header line to a non-existent header structure. In this case, pass a value of zero to `PMDFaddHeaderLine` for the **header** argument. `PMDFaddHeaderLine` will allocate and initialize the header structure, add the specified header line to it, and then return the address of the header structure in the **header** argument.

Neither of these routines actually returns the structure itself but merely a pointer to the structure (*e.g.*, the address in memory of the structure). This pointer can then be passed to the other header routines. When you are done using a header, it should be disposed of with `PMDFdisposeHeader`. This releases the memory allocated to the structure.

The header structure is an array of pointers to header line structures whose format is described below. Each entry in the array describes a particular type of header line. The `HL_` constants defined in the API include files are indices into this array.⁹ For instance, suppose that the message header of Example 1-1 is read with `PMDFreadHeader` and stored in a header structure pointed at by the pointer variable `HEADER`. Then the header structure would appear as shown in Figure 1-1.

After reading in a message header with `PMDFreadHeader`, a program can “probe” to see which header lines were specified in that message header. This is done by checking to see if the corresponding entry in the header structure array is zero (null) or not. For instance, if `HEADER[HL_REPLY_TO]` is zero, then no `Reply-to:` header line was present in the message header. From C, the *i*th entry in a header structure would be referenced using the syntax `(*hdr)[i]`; *e.g.*,

```
(*hdr)[HL_DATE]->line
```

From Pascal, use `hdr^[i]`; *e.g.*,

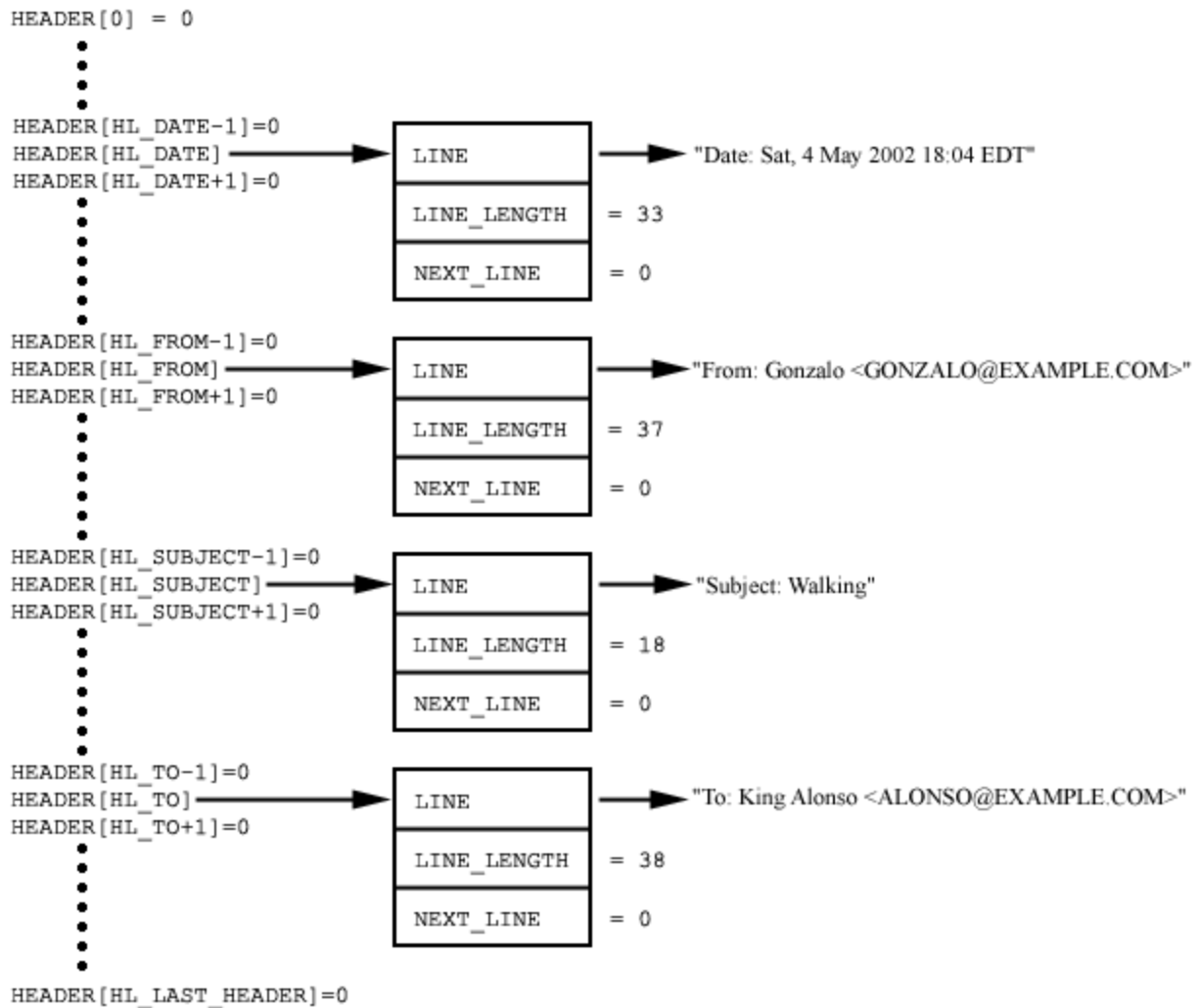
```
hdr^[HL_DATE]^line
```

The routine `display_header_lines` in Examples 1-8 and 1-9 illustrates how to walk through a header structure.

The format of a header line structure is shown in Figure 1-1. (There are actually four header line structures shown in that figure.) Each header line structure has three fields, which are as follows:

⁹ These constants are defined in the files `apidef.h` and `apidef.pen`. The C header file `apidef.h` is located in the `PMDF_COM:` directory on OpenVMS and the `/pmdf/include` directory on UNIX and NT. The Pascal environment file `apidef.pen` is located in the `PMDF_EXE:` directory on OpenVMS.

Figure 1-1 Sample Header Structure



LINE

A pointer to the header line. The header line is a null-terminated character string of length `LINE_LENGTH` bytes. The quotes shown surrounding each header line in Figure 1-1 are not part of the header line; they are used to indicate that a character string is being depicted.

LINE_LENGTH

A signed longword (4 bytes) containing the length of the character string pointed at by `LINE`. This length does not include the null terminator at the end of the string.

NEXT_LINE

A pointer to any additional header line structures describing header lines of the same type. When zero, indicates that no additional header lines of the same type exist.

The PMDF API

Message Header Structures

The `NEXT_LINE` field is the mechanism which enables more than one header line of a given type (*e.g.*, `Received:`, `Comments:`, `Keywords:`, *etc.*) to be stored in a header structure. For instance, if two `Comments:` header lines are stored in a header structure, then the first one is accessed with

```
HEADER[HL_COMMENTS]->LINE
```

and the second one with

```
HEADER[HL_COMMENTS]->NEXT_LINE->LINE
```

Each entry in the header structure array is actually a pointer to a linked list of header lines. Each header line in a given list is of the type corresponding to the index in the header structure array. That is, each header line in the linked list `HEADER[HL_x]` is of type `HL_x`. The order in which the header lines appear in the linked list are the order in which they were added to the header structure: the first header line in the list is the first one added, the second one is the second one added, and so on. Header lines added to a structure by `PMDFreadLine` are added in the order that they are read from a message header. Thus, the first `Received:` line in a header is the first one added by `PMDFreadLine`, the second `Received:` line the second one added, and so forth.

1.7 Programs that Run Indefinitely

Special attention must be given to programs that can run indefinitely. An example of such a program might be a server that continually listens for incoming mail connections and enqueues any received mail to PMDF. The following discussion is concerned with such programs. Programs which run and merely submit a few messages, loop over a queue of messages and then exit, or user interfaces should not take the steps described in this section.^a

When `PMDFinitialize` is called, site-specific configuration information is loaded. The life span of this information usually far exceeds the running time of a program that uses it. However, this isn't the case for a program that can run for weeks or months. When PMDF configuration information changes, these programs need to be made aware of the change so that they can reload this information. Subsequent calls to `PMDFdone` and `PMDFinitialize` will not accomplish this task: a program must exit and restart in order to ensure that all configuration information is reloaded.

Also, a program which enqueues or dequeues messages will open the queue cache database and possibly the PMDF log file, `mail.log_current`. Care must be taken to ensure that these files are not left open during periods of inactivity. Leaving these files open might block activities that require exclusive access to those files. Programs which run indefinitely enqueueing or dequeuing messages should always call `PMDFcloseQueueCache`, `PMDFcloseLogFile`, and, if doing message dequeue activity, `PMDFdequeueEnd` before going idle. The queue cache and log file will be automatically reopened when needed. The queue cache should not be closed while in the middle of dequeue processing; *i.e.*, `PMDFcloseQueueCache` should not be called while looping over messages in a message queue with `PMDFgetMessage`. `PMDFcloseQueueCache` should

^a User interfaces should specify `false (0)` for the `ischannel` argument to `PMDFinitialize`.

be called after `PMDFgetMessage` has returned a `PMDF__EOF` status and before again calling `PMDFdequeueInitialize`.

1.7.1 OpenVMS Considerations

The `PMDF RESTART` command is used after a change to the `PMDF` configuration to restart components of `PMDF` which run indefinitely. In addition, the `PMDF CACHE/CLOSE` command is used to force components of `PMDF` to close the queue cache database should they have it open. One such component of `PMDF` is `BN_SLAVE`. This component is a slave channel program which runs as a detached process. It starts running at system startup and continues to run, processing incoming `BITNET` mail, until `Jnet` or the system is shut down. When configuration changes are made or the queue cache needs to be rebuilt, a `PMDF RESTART` command is issued to inform `BN_SLAVE` of this fact. `BN_SLAVE` then either exits and restarts or closes the queue cache database at its earliest convenience.

The routine `PMDFsetCallBack` provides a communication path whereby a running program can be notified when a `PMDF RESTART`, `PMDF SHUTDOWN`, or `PMDF CACHE/CLOSE` command has been issued. When a `RESTART` or `SHUTDOWN` command is issued, a running program should note this fact and, as soon as is convenient, exit and restart or simply exit. In response to a `CACHE/CLOSE` command, the queue cache should be closed as soon as it is convenient to do so. This is accomplished with the `PMDFcloseQueueCache` routine. The cache will be reopened automatically when it is again needed. This is generally done by `PMDFenqueueMessage` and `PMDFgetMessage`. The queue cache should not be closed while in the middle of dequeue processing; *i.e.*, it should not be called while looping over messages in a message queue. `PMDFcloseQueueCache` should be called after `PMDFgetMessage` has returned a `PMDF__EOF` status and before again calling `PMDFdequeueInitialize`.

On `OpenVMS` systems, the communication path established by `PMDFsetCallBack` is implemented using cluster-wide resource locks. Thus, `PMDF RESTART` and `PMDF SHUTDOWN` commands issued anywhere on an `OpenVMS` cluster will be seen by all users of `PMDFsetCallBack` throughout the cluster.

1.7.2 UNIX Considerations

On `UNIX` systems, the `PMDFsetCallBack` facility is non-functional. Calls to it will merely return `PMDF__OK` without doing anything. Likewise for the `PMDFcancelCallBack` routine.

Also on `UNIX` systems, the `pmdf restart` and `shutdown` commands cannot be used to restart or shutdown site-supplied API clients. Such clients must supply their own mechanism for being signalled to either restart or shutdown.

The PMDF API

Writing Output from a Channel Program

1.8 Writing Output from a Channel Program

The `stdin`, `stdout`, and `stderr` I/O destinations (`SYS$INPUT`, `SYS$OUTPUT`, and `SYS$ERROR`) are all controlled by PMDF and will vary depending upon the context under which a channel program has been invoked. As such, programs which will operate as PMDF channels should use the `PMDFlog` routine to write information to their log file. Such programs should never write output directly to `stdout` or `stderr` or other generic I/O destinations (e.g., Pascal's "output" or FORTRAN's default output logical unit). There's no telling where such output might go: it might go to the job controller's log file, it might even go down a network pipe to a remote client or server.

Note that the channel log file is a different file than the PMDF log file; the `PMDFlog` and `PMDFcloseLogFile` are unrelated routines.

1.9 Debugging Programs and Logging Messaging Activity

The API does provide some limited debugging facilities which can help in tracking down unusual behavior. The routine `PMDFdebug` can be called to enable debugging output for either enqueueing or dequeuing operations. On OpenVMS systems, all debugging output is written to `PMDF_OUTPUT` if defined or to `SYS$OUTPUT` otherwise. On UNIX systems, debugging output is written to `stdout`.

`PMDFdebug` should be called after either `PMDFenqueueInitialize` or `PMDFdequeueInitialize` or both have been called.

Further debugging output can be enabled by setting `OS_DEBUG=1` in the PMDF option file.

Message enqueue and dequeue activities performed through the PMDF API (and callable `SEND`) will be logged when the channels involved are marked with the logging channel keyword.

1.10 Required Privileges

As should not be surprising, use of the PMDF API requires privileges. Indeed, were privileges not required, then anyone could read messages out of PMDF's message queues and send fraudulent mail messages.

1.10.1 OpenVMS Systems

Dequeuing messages only requires privileges sufficient to open, read from, and write to the queue cache database and to open, read from, rename, and delete files in the PMDF message queue directories. Under OpenVMS, the queue cache database and the queue directories are protected (`s:rwed,o:rwed,g,w`) with the files owned by the PMDF

account if one was created when PMDF was installed or owned by the SYSTEM account otherwise.

Enqueuing messages requires privileges sufficient to create, open, read from, and write to the queue cache database as well as to create subdirectories and files in the PMDF message queue directories. In addition, under OpenVMS the SYSPRV and CMKRNL privileges are required so that PMDF can submit any processing jobs required to handle an enqueued message. Note that PMDF itself does not use these privileges: they are required by the \$SNDJBC system service call used to dispatch processing jobs.

Under OpenVMS, use of the PMDFsetCallback routine requires SYSLCK privilege: cluster-wide resource locks with blocking AST's are used to signal, across a cluster, whether or not the PMDF queue cache needs to be closed and if PMDF detached processing jobs (e.g., BN_SLAVE) should exit and restart.

1.10.2 UNIX Systems

On UNIX systems, a program which will be enqueueing or dequeuing messages from or to PMDF must be owned by the account `pmdf` and run by that account. If the program is to be run by users other than `pmdf`, then it must have the `setuid` attribute.

1.11 Compiling and Linking Programs

OpenVMS Systems

To declare the API routines, data structures, HL_ constants, PMDF item codes, and PMDF error codes, C programs should use the `PMDF_COM:apidef.h` header file and Pascal programs should use the environment file `PMDF_EXE:apidef.pen`.

Linking programs to the API is accomplished with a link command of the form

```
$ LINK program, PMDF_EXE:pmdfshr_link.opt/OPT
```

where *program* is the name of the object file to link.

1.12 Examples of Using the API Routines

Several example programs, written in Pascal and C, are provided in this section:

- Examples 1-2, 1-3, and 1-4 illustrate message enqueueing;
- Examples 1-5, 1-6, and 1-7 illustrate message dequeuing;
- Examples 1-8 and 1-9 illustrate a program which both dequeues and enqueues messages; and

The PMDF API

Examples of Using the API Routines

- Examples 1–10, 1–11, and 1–12 illustrate a program which dequeues and returns all messages in its message queue.

The example routines shown in this section can be found, on OpenVMS systems, in the directory `PMDF_ROOT:[DOC.EXAMPLES]`. On UNIX systems, the examples can be found in the `/pmdf/doc/examples` directory.

Note: The example Pascal programs are intended for use on OpenVMS. To use them on UNIX or NT, changes to the examples will be required.

1.12.1 Enqueuing a Simple Message

The programs shown in Examples 1–2 and 1–3 demonstrate how to enqueue a simple “Hello world” message. The “From:” address associated with the message is that of the process running the program; the “To:” address is the local SYSTEM account. The output of these programs is given in Example 1–4. The callouts shown in the first two examples produce the corresponding output shown in the third example.

Example 1–2 Enqueuing a Message (Pascal)

```
(* api_example1.pas -- Send a "Hello world!" message to SYSTEM *)
[inherit ('pmdf_exe:apidef')] program example1;
  type uword = [word] 0..65535;
  var
    nq_context : PMDF_nq;
    user       : packed array [1..ALFA_SIZE] of char;
    user_len   : uword;
function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;
procedure check (status : integer);
  begin (* check *)
    if not odd (status) then begin
      if nq_context <> nil then PMDF_abort_message (nq_context);
      SYS$EXIT (status);
    end; (* if *)
  end; (* check *)
```

Example 1–2 Cont'd on next page

Example 1–2 (Cont.) Enqueuing a Message (Pascal)

```
begin (* example1 *)
  nq_context := nil;
  check (PMDF_initialize (false));
  check (PMDF_get_user_name (user, user_len));
  check (PMDF_enqueue_initialize);
  check (PMDF_start_message_envelope (nq_context, 'l',
                                     substr (user, 1, user_len))); ❶
  check (PMDF_add_recipient (nq_context, 'system', 'system')); ❷
  check (PMDF_start_message_header (nq_context));
  check (PMDF_write_from (nq_context, substr (user, 1, user_len))); ❸
  check (PMDF_write_date (nq_context)); ❹
  check (PMDF_write_subject (nq_context, 'Hello world!')); ❺
  check (PMDF_start_message_body (nq_context));
  check (PMDF_write_line (nq_context, 'Hello')); ❻
  check (PMDF_write_line (nq_context, ' world!')); ❼
  check (PMDF_enqueue_message (nq_context));
  check (PMDF_done);
end. (* example1 *)
```

Example 1–3 Enqueuing a Message (C)

```
/* api_example2.c -- Send a "Hello world!" message to SYSTEM */
#include <stdlib.h>

#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif

PMDF_nq *nq_context = 0;

void check (int stat)
{
  if (!(1 & stat)) {
    if (nq_context) PMDFabortMessage (&nq_context);
    exit (stat);
  }
}
```

Example 1–3 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1–3 (Cont.) Enqueuing a Message (C)

```
main ()
{
    char user[ALFA_SIZE+1];
    int user_len = ALFA_SIZE;

    check (PMDFinitialize (0));
    check (PMDFgetUserName (user, &user_len));
    check (PMDFenqueueInitialize ());
    check (PMDFstartMessageEnvelope (&nq_context, "l", 1, user, user_len)); ❶
    check (PMDFaddRecipient (&nq_context, "system", 6, "system", 6)); ❷
    check (PMDFstartMessageHeader (&nq_context));
    check (PMDFwriteFrom (&nq_context, user, user_len)); ❸
    check (PMDFwriteDate (&nq_context)); ❹
    check (PMDFwriteSubject (&nq_context, "Hello world!", 12)); ❺
    check (PMDFstartMessageBody (&nq_context));
    check (PMDFwriteLine (&nq_context, "Hello", 5)); ❻
    check (PMDFwriteLine (&nq_context, " world!", 8)); ❼
    check (PMDFenqueueMessage (&nq_context));
    check (PMDFdone ());
}
```

Example 1–4 Output of Examples 1–2 and 1–3

```
Received: from EXAMPLE.COM by EXAMPLE.COM (PMDF #1339) id
<01GP37SOPRW0A9KZFV@EXAMPLE.COM>; Sat, 4 May 2012 18:04:00 EDT
Date: 4 May 2012 18:04:00 -0400 (EDT) ❹
From: STEPHANO@EXAMPLE.COM ❸
Subject: Hello world! ❺
To: system@EXAMPLE.COM ❷
Message-id: <01GP37SOPRW2A9KZFV@EXAMPLE.COM>
X-Envelope-to: system
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT

Hello ❻
world! ❼
```

1.12.2 Dequeuing Messages

Each of the two programs shown in Examples 1–5 and 1–6 constitutes a PMDF-to-batch-SMTP channel which reads messages from a message queue, converting each message to a batch SMTP format stored in a file on disk. If the conversion is successful, then the message is dequeued and deferred otherwise. Sample output is given in Example 1–7.

The PMDF API

Examples of Using the API Routines

Note that these example programs always attempt to specify an envelope id in the batch SMTP message they output. This is done for illustration purposes only. In general, the code should check to see if the envelope id obtained with `PMDFgetEnvelopeId` is of zero length. Only if it has non-zero length should it then be outputting an envelope id.

Note: It is important to remember to define the `PMDF_CHANNEL` logical (OpenVMS) or environment variable (UNIX) to be the name of the channel (in lower case) to be serviced by this program. Also, if experimenting from your own account, do not leave this logical or environment variable defined while not experimenting — PMDF can see it when you send mail and submit that mail as though it was enqueued by the channel given by `PMDF_CHANNEL`. (This is a debugging feature.)

The following items of note are identified with callouts in each of the two programs:

- ❶ In the event of an error, the current message being processed is deferred and the program exits.
- ❷ `get_message` is a routine which will return true (1) if `PMDFgetMessage` successfully accesses a message or false (0) otherwise. If `PMDFgetMessage` returns any error other than `PMDF__EOF`, then the check routine, ❶, is invoked.
- ❸ `read_line` is a routine which will return true (1) if `PMDFgetline` successfully reads a line from a message or false (0) otherwise. If `PMDFgetline` returns any error other than `PMDF__EOF`, then the check routine, ❶, is invoked.
- ❹ `open_outbound` is a routine which opens an output file to which to write the batch SMTP command. Output from `PMDFgetUniqueString` is used in generating the file name.
- ❺ `notify` is a routine which, builds an RFC 1891 NOTIFY= parameter based upon the NOTARY flags for an envelope "To:" recipient.
- ❻ `PMDFinitialize` is invoked with the **ischannel** argument true.
- ❼ `PMDFdequeueInitialize` creates and initializes a message dequeue context.
- ❽ Using the `get_message` routine, the program loops over all messages to be processed.
- ❾ Using `PMDFgetRecipient`, the program loops over the envelope "To:" address list in the currently accessed message.
- ❿ The NOTARY flags for the current envelope "To:" recipient are obtained with `PMDFgetRecipientFlags`.
- ⓫ The disposition of the envelope "To:" recipient address is passed back to PMDF.
- ⓬ Using the `read_line` routine, the program loops over the message header and body, copying each line to the batch SMTP file.
- ⓭ Processing was successful; the processed message is dequeued.
- ⓮ All done processing messages; dispose of the message dequeue context.

Example 1–5 Message Dequeuing (Pascal)

Example 1–5 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1-5 (Cont.) Message Dequeuing (Pascal)

```
(* api_example3.pas -- Dequeue a message and output it in batch SMTP format *)
[inherit ('pmdf_exe:apidef')] program api_example3 (output);

type
  uword      = [word] 0..65535;
  string     = packed array [1..ALFA_SIZE] of char;
  bigstring  = packed array [1..BIGALFA_SIZE] of char;
  vstring    = varying [64] of char;

var
  dq_context      : PMDF_dq;
  empty           : varying [1] of char;
  env_id, from_adr, host, orig_adr, to_adr : string;
  env_id_len, from_adr_len, host_len,
  orig_adr_len, to_adr_len, txt_len      : uword;
  nflags, stat                             : integer;
  outbound_open : boolean;
  outfile       : text;
  txt           : bigstring;

procedure check (stat : integer); ❶

  var reason : varying [20] of char;

  begin (* check *)
    if not odd (stat) then begin
      writev (reason, 'Error ', stat:0);
      if dq_context <> nil then PMDF_defer_message (dq_context, true, reason);
      if outbound_open then
        close (file_variable := outfile, disposition := delete);
      halt;
    end; (* if *)
  end; (* check *)

function get_message : boolean; ❷

  var msg_file : string; msg_file_len : uword;

  begin (* get_message *)
    stat := PMDF_get_message (dq_context, msg_file, msg_file_len,
                              from_adr, from_adr_len);
    get_message := odd (stat);
    if (not odd (stat)) and (stat <> PMDF__EOF) then check (stat);
  end; (* get_message *)

function read_line : boolean; ❸

  begin (* read_line *)
    stat := PMDF_read_line (dq_context, txt, txt_len);
    read_line := odd (stat);
    if (not odd (stat)) and (stat <> PMDF__EOF) then check (stat);
  end; (* read_line *)
```

Example 1-5 Cont'd on next page

Example 1-5 (Cont.) Message Dequeuing (Pascal)

```
procedure open_outbound; ④
var
  str      : string;
  str_len  : uword;

begin (* open_outbound *)
  check (PMDF_get_unique_string (str, str_len));
  open (file_variable := outfile,
        file_name := 'ZZ' + substr (str, 1, str_len) + '.00',
        history := NEW, record_length := 1024);
  stat := status (outfile);
  if stat >= 0 then begin
    rewrite (outfile);
    outbound_open := true;
  end else begin
    writeln ('Pascal file error ', stat:0, '; aborting');
    check (0);
  end; (* if *)
end; (* open_outbound *)

function notify (nflags : integer) : vstring; ⑤
var str : vstring;

procedure add (bit : integer; toadd : varying [len] of char);
begin (* add *)
  if 0 <> uand (nflags, bit) then begin
    if length (str) = 0 then str := ' NOTIFY=' else str := str + ',';
    str := str + toadd;
  end; (* if *)
end; (* add *)

begin (* notify *)
  str := '';
  add (PMDF_RECEIPT_NEVER,      'NEVER');
  add (PMDF_RECEIPT_FAILURES,  'FAILURE');
  add (PMDF_RECEIPT_DELAYS,    'DELAY');
  add (PMDF_RECEIPT_SUCCESSES, 'SUCCESS');
  notify := str;
end; (* notify *)

begin (* api_example3 *)
  empty      := '';
  dq_context := nil;
  outbound_open := false;
  check (PMDF_initialize (true)); ⑥
  check (PMDF_get_host_name (host, host_len));
  check (PMDF_dequeue_initialize (dq_context)); ⑦
```

Example 1-5 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1-5 (Cont.) Message Dequeuing (Pascal)

```
while get_message do begin ⑧
  check (PMDF_get_envelope_id (dq_context, env_id, env_id_len));
  open_outbound;
  writeln (outfile, 'EHLO ', substr (host, 1, host_len));
  writeln (outfile, 'MAIL FROM:<', substr (from_adr, 1, from_adr_len), '>',
    ' ENVID=', substr (env_id, 1, env_id_len));
  while odd (PMDF_get_recipient (dq_context, to_adr, to_adr_len, ⑨
    orig_adr, orig_adr_len)) do begin
    check (PMDF_get_recipient_flags (dq_context, nflags)); ⑩
    writeln (outfile, 'RCPT TO:<', substr (to_adr, 1, to_adr_len), '>',
      ' ORCPT=', substr (orig_adr, 1, orig_adr_len),
      notify (nflags));
    check (PMDF_recipient_disposition (dq_context, nflags, ⑪
      PMDF_DISP_DELIVERED,
      substr (to_adr, 1, to_adr_len),
      substr (orig_adr, 1, orig_adr_len), empty));

  end; (* while *)
  writeln (outfile, 'DATA');
  while read_line do begin ⑫
    if txt_len > 0 then if txt[1] = '.' then write (outfile, '.');
    writeln (outfile, substr (txt, 1, txt_len));
  end; (* while *)
  writeln (outfile, '.');
  writeln (outfile, 'QUIT');
  close (outfile); outbound_open := false;
  check (PMDF_dequeue_message_end (dq_context, false, empty)); ⑬
end; (* while *)
check (PMDF_dequeue_end (dq_context)); ⑭
check (PMDF_done);
end. (* api_example3 *)
```

Example 1-6 Message Dequeuing (C)

```
/* api_example4.c -- Dequeue a message and output it in batch SMTP format */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif
typedef char string[ALFA_SIZE+1];
```

Example 1-6 Cont'd on next page

Example 1–6 (Cont.) Message Dequeuing (C)

```
string filename, from_adr, txt;
int outbound_open, txt_len;
PMDF_dq *dq_context = 0;
FILE *outfile;

void check (int stat) ❶
{
    char reason[20];
    if (!(1 & stat)) {
        sprintf (reason, "Reason %d", stat);
        if (dq_context) PMDFdeferMessage (&dq_context, 1, reason, strlen (reason));
        if (outbound_open) {
            fclose (outfile);
            remove (filename);
        }
        if (!stat) exit (0);
        else exit (stat);
    }
}

int get_message (void) ❷
{
    string msg_file;
    int from_adr_len, msg_file_len, stat;

    from_adr_len = msg_file_len = ALFA_SIZE;
    stat = PMDFgetMessage (&dq_context, msg_file, &msg_file_len,
                          from_adr, &from_adr_len);
    if (!(1 & stat) && stat != PMDF__EOF) check (stat);
    return (1 & stat);
}

int read_line (void) ❸
{
    int stat;

    txt_len = BIGALFA_SIZE;
    stat = PMDFreadLine (&dq_context, txt, &txt_len);
    if ( !(1 & stat) && stat != PMDF__EOF) check (stat);
    return (1 & stat);
}

void open_outbound (void) ❹
{
    char str[18+5+1];
    int str_len = 18;
}
```

Example 1–6 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1–6 (Cont.) Message Dequeuing (C)

```
check (PMDFgetUniqueString (str, &str_len));
sprintf (filename, "ZZ%s.00", str);
outfile = fopen (filename, "w");
if (!outfile) {
    fprintf (stderr, "Error opening output file; aborting\n", filename);
    check (0);
}
outbound_open = 1;
}

void add (int nflags, int bit, char *src, char *dst)
{
    if (!(nflags & bit)) return;
    if (*dst) strcat (dst, ",");
    strcat (dst, src);
}

void make_notify (int nflags, char *buf) ⑤
{
    *buf = '\0';
    add (nflags, PMDF_RECEIPT_NEVER,      "NEVER",   buf);
    add (nflags, PMDF_RECEIPT_FAILURES,   "FAILURE",  buf);
    add (nflags, PMDF_RECEIPT_DELAYS,     "DELAY",    buf);
    add (nflags, PMDF_RECEIPT_SUCCESSES,  "SUCCESS",  buf);
}

main ()
{
    string env_id, host, orig_adr, to_adr;
    int env_id_len, nflags, host_len, orig_adr_len, to_adr_len;
    char notify[64];

    outbound_open = 0;
    check (PMDFinitialize (1)); ⑥
    host_len = ALFA_SIZE;
    check (PMDFgetHostName (host, &host_len));
    check (PMDFdequeueInitialize (&dq_context)); ⑦
}
```

Example 1–6 Cont'd on next page

Example 1–6 (Cont.) Message Dequeuing (C)

```
while (get_message ()) { ❸
    env_id_len = ALFA_SIZE;
    check (PMDfgetEnvelopeId (&dq_context, env_id, &env_id_len));
    open_outbound ();
    fprintf (outfile, "EHLO %s\n", host);
    fprintf (outfile, "MAIL FROM:<%s> ENVID=%s\n", from_adr, env_id);
    to_adr_len = orig_adr_len = ALFA_SIZE;
    while (1 & PMDFgetRecipient (&dq_context, to_adr, &to_adr_len, ❹
        orig_adr, &orig_adr_len))
    {
        check (PMDfgetRecipientFlags (&dq_context, &nflags)); ❺
        make_notify (nflags, notify);
        fprintf (outfile, "RCPT TO:<%s> ORCPT=%s", to_adr, orig_adr);
        if (notify[0]) fprintf (outfile, " NOTIFY=%s\n", notify);
        else fprintf (outfile, "\n");
        check (PMDfrecipientDisposition (&dq_context, nflags, ❻
            PMDF_DISP_DELIVERED, to_adr,
            to_adr_len, orig_adr, orig_adr_len,
            NULL, 0));
        to_adr_len = orig_adr_len = ALFA_SIZE;
    }
    fprintf (outfile, "DATA\n");
    while (read_line ()) { ❻
        if (txt_len > 0) if (txt[0] == '.') fprintf (outfile, ".");
        fprintf (outfile, "%s\n", txt);
    }
    fprintf (outfile, ".\nQUIT\n");
    fclose (outfile); outbound_open = 0;
    check (PMDfdequeueMessageEnd (&dq_context, 0, NULL, 0)); ❻
}
check (PMDfdequeueEnd (&dq_context)); ❽
check (PMDfdone ());
}
```

1.12.3 Dequeuing & Re-enqueuing Messages

The programs shown in Examples 1–8 and 1–9 will loop through all messages in a message queue, converting the body of each message and re-enqueuing the converted message back to PMDF. The conversion process involves applying the “rot13” encoding used by many news readers to encode potentially offensive message content.

Note: It is important to remember to define the PMDF_CHANNEL logical (OpenVMS) or environment variable (UNIX) to be the name of the channel (in lower case) to be serviced by this program. Also, if experimenting from your own account, do not leave this logical

The PMDF API

Examples of Using the API Routines

Example 1-7 Output of Examples 1-5 and 1-6

```
EHLO EXAMPLE.COM
MAIL FROM:<stephano@example.com> ENVID=01ISXU84PB929AMHQL@EXAMPLE.COM
RCPT TO:<caliban@example.com>
  ORCPT=rfc822;caliban@island.example.com NOTIFY=FAILURES,DELAY
DATA
Received: from EXAMPLE.COM by EXAMPLE.COM (PMDf #1339) id
  <01GP3A97QW9CAATXKZ@EXAMPLE.COM>; Sat, 04 May 2012 18:04:00 EDT
Date: 04 May 2012 18:04:00 -0400 (EDT)
From: "Stephano the Drunken Butler" <stephano@example.com>
Subject: Testing
To: "Caliban the Savage" <caliban@island.example.com>
Message-id: <01GP3A97R5WIAATXKZ@EXAMPLE.COM>
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT

This is a test of the emergency broadcasting system.
Please do not be alarmed. Please do not hold your breath.

Bye
.
QUIT
```

or environment variable defined while not experimenting — PMDF can see it when you send mail and submit that mail as though it was enqueued by the channel given by PMDF_CHANNEL. (This is a debugging feature.)

The following items of note are identified with callouts in each of the two programs:

- ❶ In the event of an error, the current message being processed is deferred, any new message being enqueued is aborted, and the program exits.
- ❷ `get_message` is a routine which will return true (1) if `PMDfgetMessage` successfully accesses a message or false (0) otherwise. If `PMDfgetMessage` returns any error other than `PMDf__EOF`, then the check routine is invoked.
- ❸ `read_line` is a routine which will return true (1) if `PMDfgetLine` successfully reads a line from a message or false (0) otherwise. If `PMDfreadLine` returns any error other than `PMDf__EOF`, then the check routine is invoked.
- ❹ A routine to walk through the header structure, `hdr`, and display any header lines stored in the structure. This routine does not serve any real purpose here other than to illustrate how to walk a header structure.
- ❺ The infamous `rot13` filter.
- ❻ `PMDfinitialize` is invoked with the **`ischannel`** argument true.
- ❼ `PMDfgetChannelName` is used to determine the name of the channel being processed. This information is later passed to `PMDfstartMessageEnvelope`.
- ❽ `PMDfdequeueInitialize` creates and initializes a message dequeue context.
- ❾ Using the `get_message` routine, the program loops over all messages to be processed.
- ❿ The envelope id for the message being processed is obtained. This envelope id will be carried over to the new message which will be enqueued.

The PMDF API

Examples of Using the API Routines

- ⑪ Begin a message enqueue context. This new message will be the converted form of the message to be dequeued.
- ⑫ Set the envelope id for the new message to be that of the old message.
- ⑬ Using `PMDFgetRecipient`, the program loops over the envelope "To:" address list in an accessed message.
- ⑭ The NOTARY flags for the current envelope "To:" address are obtained with `PMDFgetRecipientFlags`. They are then copied over to the same envelope "To:" address in the new message by calling `PMDFsetRecipientFlags` and then `PMDFaddRecipient`.
- ⑮ The disposition of the envelope "To:" address is declared.
- ⑯ The envelope is ended and the message header started.
- ⑰ `PMDFreadHeader` and `PMDFwriteHeader` is used to copy, without alteration, the message header from the old message to the new message.
- ⑱ Call `display_header_lines` to display, on the terminal, the contents of the header structure, `hdr`. This is merely done as an example of walking through a header structure; displaying the structure serves no other useful purpose in this example.
- ⑲ Using the `read_line` routine, the program loops over the message body, reading each line from the original messages, converting it, and then writing it to the new message being enqueued.
- ⑳ The new message is enqueued and the message enqueue context disposed of.
- ㉑ The old message is dequeued.
- ㉒ All done processing messages; dispose of the message dequeue context;

Example 1–8 Message Dequeuing & Re-enqueuing (Pascal)

```
(* api_example5.pas -- Dequeue a message, rot13 the message body,
                        and then requeue the message *)

[inherit ('pmdf_exe:apidef')] program api_example5 (output);

type
  uword      = [word] 0..65535;
  string     = packed array [1..ALFA_SIZE] of char;
  bigstring  = packed array [1..BIGALFA_SIZE] of char;

var
  channel, env_id, from_adr, orig_adr, to_adr : string;
  channel_len, env_id_len, from_adr_len,
    orig_adr_len, to_adr_len, txt_len          : uword;
  dq_context                                : PMDF_dq;
  empty                                     : varying [1] of char;
  hdr                                       : PMDF_hdr;
  i, nflags, stat                           : integer;
  nq_context                                : PMDF_nq;
  outfile                                   : text;
  txt                                       : bigstring;
```

Example 1–8 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1-8 (Cont.) Message Dequeuing & Re-enqueuing (Pascal)

```
function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;
procedure check (stat : integer); ❶
    var reason : varying [20] of char;
    begin (* check *)
        if not odd (stat) then begin
            writev (reason, 'Reason ', stat:0);
            if dq_context <> nil then PMDF_defer_message (dq_context, true, reason);
            if nq_context <> nil then PMDF_abort_message (nq_context);
            if stat = 0 then SYS$EXIT (1) else SYS$EXIT (stat);
        end; (* if *)
    end; (* check *)
function get_message : boolean; ❷
    var msg_file : string; msg_file_len : uword;
    begin (* get_message *)
        stat := PMDF_get_message (dq_context, msg_file, msg_file_len,
            from_adr, from_adr_len);
        get_message := odd (stat);
        if (not odd (stat)) and (stat <> PMDF__EOF) then check (stat);
    end; (* get_message *)
function read_line : boolean; ❸
    begin (* read_line *)
        stat := PMDF_read_line (dq_context, txt, txt_len);
        read_line := odd (stat);
        if (not odd (stat)) and (stat <> PMDF__EOF) then check (stat);
    end; (* read_line *)
procedure display_header_lines (hdr : PMDF_hdr); ❹
    var i : integer; hdr_line : PMDF_hdr_line_ptr;
    begin (* display_header_lines *)
        for i := HL_FIRST_HEADER to HL_LAST_HEADER do begin
            if hdr^[i] <> nil then begin
                hdr_line := hdr^[i];
                while hdr_line <> nil do begin
                    writeln (substr (hdr_line^.line^, 1, hdr_line^.line_length));
                    hdr_line := hdr_line^.next_line;
                end; (* while *)
            end; (* if *)
        end; (* for *)
    end; (* display_header_lines *)
```

Example 1-8 Cont'd on next page

Example 1–8 (Cont.) Message Dequeuing & Re-enqueuing (Pascal)

```
function rot13 (c : char) : char; ⑤
begin (* rot13 *)
  if c in ['A'..'Z'] then
    rot13 := chr (((ord (c) - ord ('A') + 13) mod 26) + ord ('A'))
  else if c in ['a'..'z'] then
    rot13 := chr (((ord (c) - ord ('a') + 13) mod 26) + ord ('a'))
  else rot13 := c;
end; (* rot13 *)

begin (* api_example5 *)
  empty      := '';
  hdr        := nil;
  dq_context := nil;
  nq_context := nil;
  check (PMDF_initialize (true)); ⑥
  check (PMDF_get_channel_name (channel, channel_len)); ⑦
  check (PMDF_dequeue_initialize (dq_context)); ⑧
  check (PMDF_enqueue_initialize);
  while get_message do begin ⑨
    check (PMDF_get_envelope_id (dq_context, env_id, env_id_len)); ⑩
    check (PMDF_start_message_envelope (nq_context, ⑪
      substr (channel, 1, channel_len),
      substr (from_adr, 1, from_adr_len));
    check (PMDF_set_envelope_id (nq_context, substr (env_id, 1, env_id_len)); ⑫
    while odd (PMDF_get_recipient (dq_context, to_adr, to_adr_len, ⑬
      orig_adr, orig_adr_len)) do begin
      check (PMDF_get_recipient_flags (dq_context, nflags)); ⑭
      check (PMDF_set_recipient_flags (nq_context, nflags));
      check (PMDF_add_recipient (nq_context, substr (to_adr, 1, to_adr_len),
        substr (orig_adr, 1, orig_adr_len));
      check (PMDF_recipient_disposition (dq_context, nflags, ⑮
        PMDF_DISP_DELIVERED,
        substr (to_adr, 1, to_adr_len),
        substr (orig_adr, 1, orig_adr_len), empty));
    end; (* while *)
    check (PMDF_start_message_header (nq_context)); ⑯
    check (PMDF_read_header (dq_context, hdr)); ⑰
    display_header_lines (hdr); ⑱
    check (PMDF_write_header (nq_context, hdr));
    check (PMDF_dispose_header (hdr));
    check (PMDF_start_message_body (nq_context));
```

Example 1–8 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1-8 (Cont.) Message Dequeuing & Re-enqueuing (Pascal)

```
while read_line do begin ①⑨
  for i := 1 to txt_len do txt[i] := rot13 (txt[i]);
  check (PMDF_write_line (nq_context, substr (txt, 1, txt_len)));
end; (* while *)
check (PMDF_enqueue_message (nq_context)); ②⑩
check (PMDF_dequeue_message_end (dq_context, false, empty)); ②①
end; (* while *)
check (PMDF_dequeue_end (dq_context)); ②②
check (PMDF_done);
end. (* api_example5 *)
```

Example 1-9 Message Dequeuing & Re-enqueuing (C)

```
/* api_example6.c -- Dequeue a message, rot13 the message body,
   and then requeue the message */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif

typedef char string[ALFA_SIZE+1];

string from_adr, txt;
int from_adr_len, txt_len;
PMDF_nq *nq_context = 0;
PMDF_dq *dq_context = 0;

void check (int stat) ①
{
  char reason[20];
  if (!(1 & stat)) {
    sprintf (reason, "Reason %d", stat);
    if (dq_context) PMDFdeferMessage (&dq_context, 1, reason, strlen (reason));
    if (nq_context) PMDFabortMessage (&nq_context);
    if (!stat) exit (0);
    else exit (stat);
  }
}
```

Example 1-9 Cont'd on next page

Example 1–9 (Cont.) Message Dequeuing & Re-enqueuing (C)

```
int get_message (void) ❷
{
    string msg_file;
    int msg_file_len, stat;

    msg_file_len = from_adr_len = ALFA_SIZE;
    stat = PMDFgetMessage (&dq_context, msg_file, &msg_file_len,
                          from_adr, &from_adr_len);
    if (!(1 & stat) && stat != PMDF__EOF) check (stat);
    return (1 & stat);
}

int read_line (void) ❸
{
    int stat;

    txt_len = BIGALFA_SIZE;
    stat = PMDFreadLine (&dq_context, txt, &txt_len);
    if ( !(1 & stat) && stat != PMDF__EOF) check (stat);
    return (1 & stat);
}

void display_header_lines (PMDF_hdr *hdr) ❹
{
    int i;
    PMDF_hdr_line *hdr_line;

    for (i = HL_FIRST_HEADER; i <= HL_LAST_HEADER; i++) {
        if ((*hdr)[i]) {
            hdr_line = (*hdr)[i];
            while (hdr_line) {
                printf ("%s\n", hdr_line->line);
                hdr_line = hdr_line->next_line;
            }
        }
    }
}

char rot13 (char c) ❺
{
    if ('A' <= c && c <= 'Z') return (((c - 'A' + 13) % 26) + 'A');
    else if ('a' <= c && c <= 'z') return (((c - 'a' + 13) % 26) + 'a');
    else return (c);
}
```

Example 1–9 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1–9 (Cont.) Message Dequeuing & Re-enqueuing (C)

```
main ()
{
    string channel, env_id, orig_adr, to_adr;
    int channel_len, env_id_len, nflags, i, orig_adr_len, to_adr_len;
    PMDF_hdr *hdr;
    unsigned int key;

    check (PMDFinitialize (1)); ⑥
    channel_len = ALFA_SIZE;
    check (PMDFgetChannelName (channel, &channel_len, &key, &key)); ⑦
    check (PMDFdequeueInitialize (&dq_context)); ⑧
    check (PMDFenqueueInitialize ());
    while (get_message ()) { ⑨
        env_id_len = ALFA_SIZE;
        check (PMDFgetEnvelopeId (&dq_context, env_id, &env_id_len)); ⑩
        check (PMDFstartMessageEnvelope (&nq_context, channel, channel_len, ⑪
            from_adr, from_adr_len));
        check (PMDFsetEnvelopeId (&nq_context, env_id, env_id_len)); ⑫
        to_adr_len = orig_adr_len = ALFA_SIZE;
        while (1 & PMDFgetRecipient (&dq_context, to_adr, &to_adr_len, ⑬
            orig_adr, &orig_adr_len))
        {
            check (PMDFgetRecipientFlags (&dq_context, &nflags)); ⑭
            check (PMDFsetRecipientFlags (&nq_context, nflags));
            check (PMDFaddRecipient (&nq_context, to_adr, to_adr_len,
                orig_adr, orig_adr_len));
            check (PMDFrecipientDisposition (&dq_context, nflags, ⑮
                PMDF_DISP_DELIVERED, to_adr,
                to_adr_len, orig_adr, orig_adr_len,
                NULL, 0));
            to_adr_len = orig_adr_len = ALFA_SIZE;
        }
        check (PMDFstartMessageHeader (&nq_context)); ⑯
        check (PMDFreadHeader (&dq_context, &hdr)); ⑰
        display_header_lines (hdr); ⑱
        check (PMDFwriteHeader (&nq_context, hdr));
        check (PMDFdisposeHeader (&hdr));
        check (PMDFstartMessageBody (&nq_context));
        while (read_line ()) { ⑲
            for (i = 0; i < txt_len - 1; i++) txt[i] = rot13 (txt[i]);
            check (PMDFwriteLine (&nq_context, txt, txt_len));
        }
        check (PMDFenqueueMessage (&nq_context)); ⑳
        check (PMDFdequeueMessageEnd (&dq_context, 0, NULL, 0)); ㉑
    }
    check (PMDFdequeueEnd (&dq_context)); ㉒
    check (PMDFdone ());
}
```

1.12.4 Dequeuing & Returning Messages

Examples 1–10 and 1–11 illustrate the use of `PMDFdequeueMessageEnd` to return a message to its originator. A message in the channel's queue is accessed and each of its envelope "To:" recipients are given a disposition of `PMDF_DISP_RETURN` which indicates that the message is undeliverable for that recipient. Then, when `PMDFdequeueMessageEnd` is called, a bounce message is automatically generated and sent back to the original message's originator. The original message is then removed from the queue. Note that no notification message will be generated if the NOTARY flags for all of the recipients specify `PMDF_RETURN_NEVER`.

These two particular examples, through the use of `PMDFgetMessage`, return each and every message in a message queue. A sample returned message is shown in Example 1–12.

Note: It is important to remember to define the `PMDF_CHANNEL` logical (OpenVMS) or environment variable (UNIX) to be the name of the channel (in lower case) to be serviced by this program. Also, if experimenting from your own account, do not leave this logical or environment variable defined while not experimenting — PMDF can see it when you send mail and submit that mail as though it was enqueued by the channel given by `PMDF_CHANNEL`. (This is a debugging feature.)

The following items of note are identified with callouts in each of the two programs:

- ❶ In the event of an error, the current message being processed is deferred, any new message being enqueued is aborted, and the program exits.
- ❷ `get_message` is a routine which will return true (1) if `PMDFgetMessage` successfully accesses a message or false (0) otherwise. If `PMDFgetMessage` returns any error other than `PMDF__EOF`, then the check routine is invoked.
- ❸ `PMDFinitialize` is invoked with the **ischannel** argument true.
- ❹ Initialize a message dequeue context with `PMDFdequeueInitialize`.
- ❺ Using the `get_message` routine, the program loops over all messages to be processed.
- ❻ Obtain the next envelope "To:" address for the current message.
- ❼ Obtain the NOTARY flags for the envelope "To:" address just obtained.
- ❽ Set the disposition for this envelope "To:" address to `PMDF_DISP_RETURN`. This will cause the message to be returned as undeliverable for this envelope "To:" address.
- ❾ The message is automatically returned when `PMDFdequeueMessageEnd` is called.
- ❿ All done processing messages; dispose of the message dequeue context.

Example 1–10 Dequeuing & Returning Messages (Pascal)

Example 1–10 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1–10 (Cont.) Dequeuing & Returning Messages (Pascal)

```
(* api_example7.pas -- Return channel which returns all mail queued to it *)
[inherit ('pmdf_exe:apidef')] program api_example7;

type
  uword  = [word] 0..65535;
  string = packed array [1..ALFA_SIZE] of char;

var
  from_adr, orig_adr, to_adr           : string;
  from_adr_len, orig_adr_len, to_adr_len : uword;
  dq_context                           : PMDF_dq;
  empty                                : varying [1] of char;
  nflags                               : integer;

function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;

procedure check (stat : integer); ❶
  var reason : varying [20] of char;
begin (* check *)
  if not odd (stat) then begin
    writev (reason, 'Error ', stat:0);
    if dq_context <> nil then PMDF_defer_message (dq_context, true, reason);
  end; (* if *)
end; (* check *)

function get_message : boolean; ❷
  var msg_file : string; msg_file_len : uword; stat : integer;
begin (* get_message *)
  stat := PMDF_get_message (dq_context, msg_file, msg_file_len,
                           from_adr, from_adr_len);
  get_message := odd (stat);
  if (not odd (stat)) and (stat <> PMDF__EOF) then check (stat);
end; (* get_message *)

begin (* api_example7 *)
  dq_context := nil;
  empty      := '';
  check (PMDF_initialize (true)); ❸
  check (PMDF_dequeue_initialize (dq_context)); ❹
end;
```

Example 1–10 Cont'd on next page

Example 1-10 (Cont.) Dequeuing & Returning Messages (Pascal)

```
while get_message do begin ⑤
  while odd (PMDF_get_recipient (dq_context, to_adr, to_adr_len, ⑥
                                orig_adr, orig_adr_len)) do begin
    check (PMDF_get_recipient_flags (dq_context, nflags)); ⑦
    check (PMDF_recipient_disposition (dq_context, nflags, ⑧
                                       PMDF_DISP_RETURN, substr (to_adr, 1, to_adr_len),
                                       substr (orig_adr, 1, orig_adr_len),
                                       'Message undeliverable; returned by the postmaster'));
    end; (* while *)
    check (PMDF_dequeue_message_end (dq_context, false, empty)); ⑨
  end; (* while *)
  check (PMDF_dequeue_end (dq_context)); ⑩
  check (PMDF_done);
end. (* api_example7 *)
```

Example 1-11 Dequeuing & Returning Messages (C)

```
/* api_example8.c -- Return channel which returns all mail queued to it */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif

typedef char string[ALFA_SIZE+1];

string from_adr;
int from_adr_len, item_index;
PMDF_dq *dq_context = 0;

void check (int stat) ①
{
  char reason[20];
  if (!(1 & stat)) {
    sprintf (reason, "Reason %d", stat);
    if (dq_context) PMDFdeferMessage (&dq_context, 1, reason, strlen (reason));
    if (!stat) exit (0);
    else exit (stat);
  }
}
```

Example 1-11 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1–11 (Cont.) Dequeuing & Returning Messages (C)

```
int get_message (void) ❷
{
    string msg_file;
    int msg_file_len, stat;

    msg_file_len = from_adr_len = ALFA_SIZE;
    stat = PMDFgetMessage (&dq_context, msg_file, &msg_file_len,
                          from_adr, &from_adr_len);
    if (!(1 & stat) && stat != PMDF__EOF) check (stat);
    return (1 & stat);
}

main ()
{
    string orig_adr, to_adr;
    int i, nflags, orig_adr_len, to_adr_len;

    channel_len = ALFA_SIZE;
    check (PMDFinitialize (1)); ❸
    check (PMDFdequeueInitialize (&dq_context)); ❹
    while (get_message ()) { ❺
        item_index = 0;
        to_adr_len = orig_adr_len = ALFA_SIZE;
        while (1 & PMDFgetRecipient (&dq_context, to_adr, &to_adr_len, ❻
                                     orig_adr, &orig_adr_len)) {
            check (PMDFgetRecipientFlags (&dq_context, &nflags)); ❼
            check (PMDFrecipientDisposition (&dq_context, nflags, ❽
                                             PMDF_DISP_RETURN, to_adr, to_adr_len,
                                             orig_adr, orig_adr_len,
                                             "Message undeliverable; returned by the postmaster", 49));
            to_adr_len = orig_adr_len = ALFA_SIZE;
        }
        check (PMDFdequeueMessageEnd (&dq_context, 0, "", 0)); ❾
    }
    check (PMDFdequeueEnd (&dq_context)); ❿
    check (PMDFdone ());
}

```

Example 1–12 shows a sample return message generated by `PMDFreturnMessage`. In that example, the following items are marked with callouts: the message header, ❶; a MIME header line indicating that the message is a multi-part message, ❷; the first body part which contains a human readable explanation as to why the message was returned, ❸; the second body part which contains a machine readable explanation as to why the message was returned, ❹; and the third body part containing the message being returned, ❺.

Example 1–12 Output of Examples 1–10 and 1–11

Example 1–12 Cont'd on next page

Example 1-12 (Cont.) Output of Examples 1-10 and 1-11

```
Received: from example.com (PMDF V6.1 #8790) ❶
  id <01IXGG2X55A88Y55Z3@example.com>; Sat, 04 May 2012 18:04:00 EDT
Date: Sat, 04 May 2012 18:04:00 EDT
From: PMDF Internet Messaging <postmaster@example.com>
Subject: Delivery Notification: Delivery has been manually aborted
To: Trinculo@example.com, postmaster@example.com
Message-id: <01IXGG2Y8J468Y55Z3@example.com>
MIME-version: 1.0
Content-type: MULTIPART/REPORT; REPORT-TYPE=DELIVERY-STATUS; ❷
  BOUNDARY="Boundary_(ID_78nMbcjsTsCboulbhJC84A)"

--Boundary_(ID_78nMbcjsTsCboulbhJC84A) ❸
Content-type: text/plain; charset=us-ascii
Content-language: EN-US

This report relates to a message you sent with the following header fields:

  Message-id: <01IXGGR0TSYS8Y55Z3@example.com>
  Date: Sat, 04 May 2012 18:04:00 -0400 (EDT)
  From: Trinculo@example.com
  To: Stephano@example.com
  Subject: Meeting next Wednesday

Your message is being returned.  It was forced to return by the postmaster.
The recipient list for this message was:

  Recipient address: Stephano@example.com
  Reason: Message undeliverable; returned by the postmaster

--Boundary_(ID_78nMbcjsTsCboulbhJC84A) ❹
Content-type: message/DELIVERY-STATUS

Original-envelope-id: 01IXGFBILT3M8Y55Z3@example.com
Reporting-MTA: dns;example.com

Action: failed
Status: 5.0.0 (Message undeliverable; returned by the postmaster)
Original-recipient: rfc822;Stephano@example.com
Final-recipient: rfc822;Stephano@example.com
```

Example 1-12 Cont'd on next page

The PMDF API

Examples of Using the API Routines

Example 1–12 (Cont.) Output of Examples 1–10 and 1–11

```
--Boundary_(ID_78nMbcjsTsCboulbhJC84A) ⑤
Content-type: text/rfc822-headers

Return-path: Trinculo@example.com
Received: from example.com by example.com (PMDF V6.1 #8790)
  id <01IXGG2X55A88Y55Z3@example.com>
  (original mail from Trinculo@example.com); Sat, 04 May 2012 18:04:00 EDT
Received: from example.com by example.com (PMDF V6.1 #8790)
  id <01IXGFBKQIO8Y55Z3@example.com> for Stephano@example.com;
  Sat, 04 May 2012 18:04:00 EDT
Date: Sat, 04 May 2012 18:04:00 -0400 (EDT)
From: Trinculo@example.com
Subject: Meeting next Wednesday
To: Stephano@example.com
Message-id: <01IXGFBILT3M8Y55Z3@example.com>
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII

Can we reschedule the meeting of comic relief characters to be at 14:30?
--Boundary_(ID_78nMbcjsTsCboulbhJC84A)--
```

1.13 API Routine Descriptions

The strings passed as input to the C format API routines need not be zero terminated; the API routines ignore any zero terminators and exclusively use the associated length argument when determining the strings length. On output, however, the C format API routines will always add zero terminators to output strings as well as return the strings' lengths in the associated length arguments.

1.13.1 Summary of Routines

Table 1–1 summarizes the routines included in the PMDF API.

Table 1–1 Routines Included in the PMDF API

Enqueue routines	Description
PMDFenqueueInitialize	Prepare for one or more message enqueues
PMDFstartMessageEnvelope	Begin a message enqueue context; specify the envelope "From:" address
PMDFsetEnvelopeld	Set the envelope id for the message
PMDFsetRecipientType	Specify if an address is a "To:", "Cc:", or "Bcc:" address
PMDFsetRecipientFlags	Set NOTARY flags for next envelope recipient address
PMDFaliasNoExpansion	Inhibit expansion of aliases; generally, this routine should not be used

Table 1–1 (Cont.) Routines Included in the PMDF API

Enqueue routines	Description
PMDFaddRecipient	Specify "To:", "Cc:", and "Bcc:" addresses
PMDFstartMessageHeader	End the message envelope and begin the message header
PMDFwriteDate	Output a "Date:" header line
PMDFwriteFrom	Output a "From:" header line
PMDFwriteSubject	Output a "Subject:" header line
PMDFwriteHeader	Output a header structure
PMDFstartMessageBody	End the message header and begin the message body
PMDFwriteLine	Output a line to the message header or body
PMDFwriteText	Output a text string to the message header or body
PMDFenqueueMessage	End the message and enqueue it; dispose of the message enqueue context
PMDFabortMessage	Abort a message and dispose of the message enqueue context
PMDFreceiptControl	Control the use of delivery and receipt request headers
PMDFsetLimits	Set message size limits used to fragment messages
PMDFsetReceiptAddresses	Set delivery and read receipt request addresses
Dequeue routines	Description
PMDFdequeueInitialize	Prepare for one or more message dequeues; create a dequeue context
PMDFgetMessage	Access a queued message; return the envelope "From:" address
PMDFgetEnvelopeld	Get the message's envelope id
PMDFgetMessageId	Get the message's message id
PMDFgetRecipient	Read the next envelope "To:" address
PMDFgetRecipientFlags	Obtain NOTARY flags for previous envelope recipient address
PMDFcopyMessage	Copy the queued message to a new message being enqueued
PMDFrecipientDisposition	Specify the disposition of a recipient address.
PMDFreadHeader	Read the header of a message
PMDFreadLine	Read a line from a message; line feed record terminator is stripped by API
PMDFreadText	Read a line from a message; line feed record terminator is not stripped by API
PMDFreadFailureLog	Read a line from the message delivery failure log, if present
PMDFrewindMessage	Go back to the start of the message header
PMDFdequeueMessageEnd	Remove a message from the message queue
PMDFdequeueEnd	Dispose of a message dequeue context
Address parsing	Description
PMDFaddressParseList	Parse a list of address producing an address context
PMDFaddressGet	Extract an individual address from a list of parsed addresses
PMDFaddressGetProperty	Extract a property of an individual address from a list of parsed addresses
PMDFaddressDispose	Dispose of an address context
PMDFgetAddressProperty	Parse an address and return the specified property
Option file processing	Description
PMDFoptionDispose	Dispose of an option file context

The PMDF API

API Routine Descriptions

Table 1–1 (Cont.) Routines Included in the PMDF API

Option file processing	Description
PMDFOptionGetInteger	Obtain the value associated with an integer-valued option
PMDFOptionGetReal	Obtain the value associated with a real-valued option
PMDFOptionGetString	Obtain the value associated with a string-valued option
PMDFOptionRead	Process an option file
Miscellaneous routines	Description
PMDFAbortProgram	Abort the currently running program
PMDFAddHeaderLine	Add a header line to a header structure
PMDFAddressToChannel	Return the name of the channel to which the specified address rewrites
PMDFCancelCallBack	Cancel any call backs
PMDFchannelToHost	Return the official host name associated with a channel
PMDFcloseLogFile	Close the PMDF log file
PMDFcloseQueueCache	Close the queue cache database
PMDFdebug	Set enqueue and dequeue debugging flags
PMDFdatabaseAddEntry	Add an entry to a database
PMDFdatabaseClose	Close a database
PMDFdatabaseDeleteEntry	Remove an entry from a database
PMDFdatabaseGetEntry	Lookup an entry in a database
PMDFdeleteHeaderLine	Remove a header line from a header structure
PMDFdisposeChannelCounters	Dispose of a list of channel counters
PMDFdisposeHeader	Dispose of a message header structure
PMDFdone	Deallocate PMDF data structures and resources
PMDFgetBlockSize	Obtain the size in bytes of a PMDF block
PMDFgetChannelName	Obtain the current channel name
PMDFgetChannelCounters	Obtain channel counters
PMDFgetErrorText	Obtain information about a recent error message
PMDFgetDateTime	Obtain the current date and time
PMDFgetHostName	Obtain the official local host name
PMDFgetPostmasterAddress	Obtain the local postmaster's address
PMDFgetUniqueString	Obtain a unique string suitable for use in filenames
PMDFgetUserName	Obtain the current user name
PMDFhostToChannel	Return the name of the channel associated with the specified host name
PMDFinitialize	Initialize PMDF data structures and resources
PMDFlog	Write a line of text to a channel log file
PMDFmappingApply	Map a string with a mapping table
PMDFmappingLoad	Load a mapping table
PMDFqueueCacheEnd	Dispose of a queue cache context created with PMDFqueueCacheGetEntry
PMDFqueueCacheGetEntry	Retrieve an entry from the queue cache database
PMDFsetCallBack	Establish a call back routine
PMDFsetMutex	Provide mutex handling routines
Obsolete routines	Description
PMDFdequeueMessage	Remove a message from the message queue; superseded by PMDFrecipientDisposition and PMDFdequeueMessageEnd
PMDFdeferMessage	Defer a message for later reprocessing; superseded by PMDFrecipientDisposition and PMDFdequeueMessageEnd

Table 1–1 (Cont.) Routines Included in the PMDF API

Obsolete routines	Description
PMDFreturnMessage	Return a message as undeliverable; non-NOTARY style format; superseded by PMDFrecipientDisposition

1.13.2 Order Dependencies

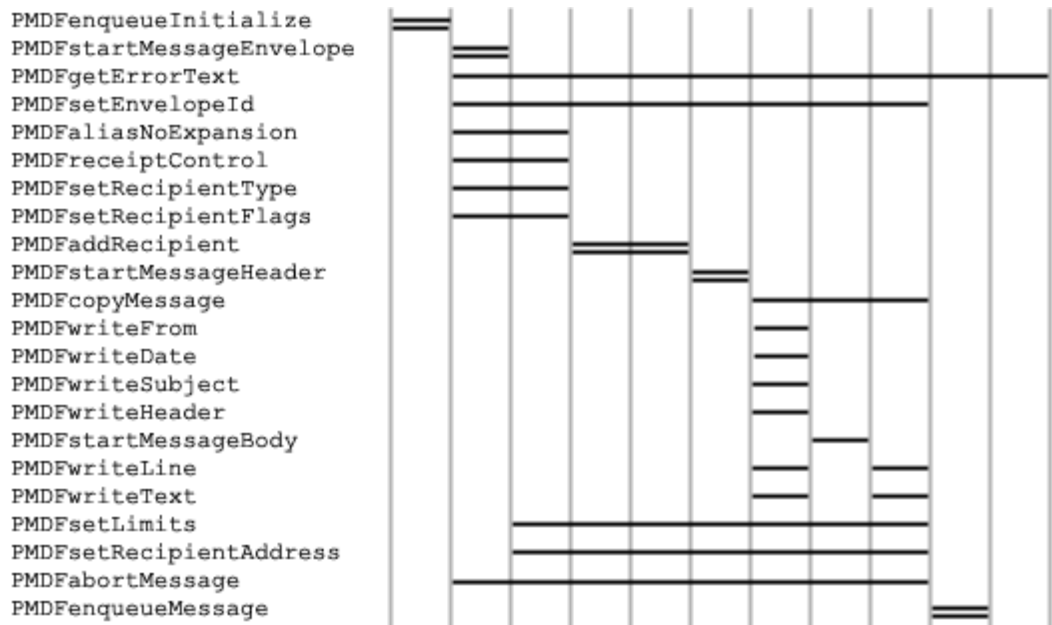
Figure 1–2 visually depicts the calling order dependency of the message enqueue routines. To the right of each routine name appears a horizontal line segment, possibly broken across a column (*e.g.*, PMDFwriteLine, PMDFwriteText). Routines for which two horizontal line segments, one atop the other, appear are required routines — routines which must be called in order to enqueue a message. These routines are PMDFenqueueInitialize, PMDFstartMessageEnvelope, PMDFaddRecipient, PMDFstartMessageHeader, and PMDFenqueueMessage. Now, to determine at which point a routine can be called, start in the leftmost column and work towards the rightmost column. Any routine whose line segment lies in the first (leftmost) column can be called first. Any routine whose line segment falls in the second column can next be called after which any routine whose line segment falls in the third column can be called, *etc.*, *etc.*. When more than one routine appears in the same column, any or all of those routines can be called in any order. Progression from left to right across the columns is mandated by the need to call the required routines. Note that of the required routines, only PMDFaddRecipient can be called multiple times for a given message.

It is assumed in Figure 1–2 that PMDFinitialize is first called before any other API routines. If more than one message is to be enqueued, PMDFenqueueInitialize should only be called once, at the start of the first message.

The PMDF API

API Routine Descriptions

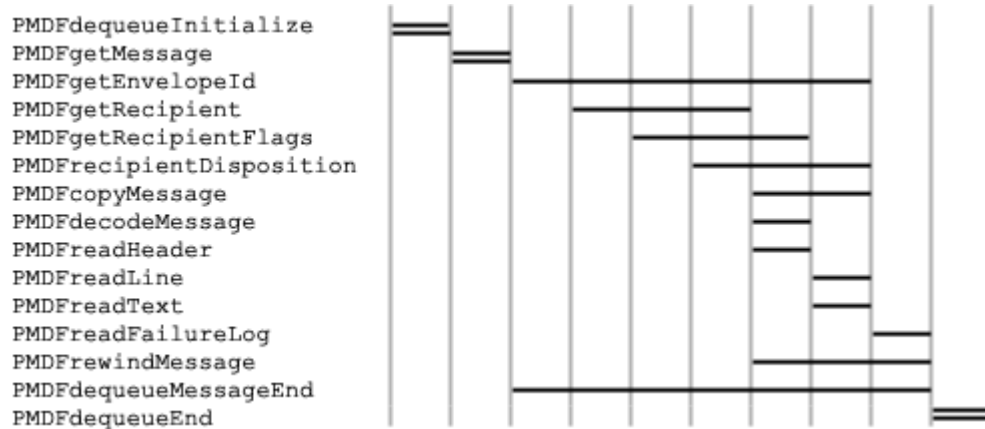
Figure 1–2 Calling Precedence for the API Message Enqueue Routines



Similarly, Figure 1–3 visually depicts the calling order dependency of the message dequeue routines. In that figure, the required routines are `PMDFdequeueInitialize`, and `PMDFgetMessage`.

In Figure 1–3, it is assumed that `PMDFinitialize` is first called before any other API routines. If more than one message is to be dequeued, `PMDFdequeueInitialize` should only be called once, at the start of the first message. `PMDFgetMessage` should be called repeatedly until the status code `PMDF__EOF` is returned at which point there are no more messages to be processed. Note that after calling `PMDFrewindMessage`, the message is rewound to the start of the message header and `PMDFreadHeader` can again be called (*i.e.*, you're back in the sixth column counting from the left).

Figure 1–3 Calling Precedence for the API Message Dequeue Routines



1.13.3 Strings Passed To and From the API

As mentioned previously, the API presents two call formats: one which uses OpenVMS-style string descriptors and another which uses C's style of passing a pointer to a string. For multi-platform code, use the C style interface.

When using the C-style interface, strings passed in need not be zero terminated: the length of the string is always determined from an associated argument specifying the length of the string. When a string is passed in which will be written to, on output the string is always zero terminated and its length, not including the zero terminator, returned in an associated length argument. On input, this length argument must give the maximum length of the string, not including the space used by a zero terminator.

Although strongly discouraged, the VMS-style interface which uses OpenVMS string descriptors can be used on UNIX as well as OpenVMS. Under UNIX the DSC\$B_DTYPE and DSC\$B_CLASS fields of descriptors are ignored by the API; all descriptors are treated as static character string descriptors (DSC\$B_CLASS = DSC\$K_CLASS_S; DSC\$B_DTYPE = DSC\$K_DTYPE_T).

There are basic string sizes used by the API. Their symbolic names and their values in PMDF V6.1 are shown in Table 1–2. As these sizes are subject to change, programmers are encouraged to use the constants defined in the supplied include files described in Section 1.11.

The PMDF API

API Routine Descriptions

Table 1–2 String Size Constants Used by the API

Symbolic name	Value
ALFA_SIZE	252
BIGALFA_SIZE	1024
CHANLENGTH	32
DATA_LENGTH	80
KEY_LENGTH	32
LONG_DATA_LENGTH	ALFA_SIZE
LONG_KEY_LENGTH	80
SHORTALFA_SIZE	40

1.13.4 Routine Descriptions

This section documents the PMDF API routines.

PMDFabortMessage

Abort a message enqueue context.

PASCAL *status* = **PMDF_abort_message** (*nq_context*)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference

C *status* = **PMDFabortMessage** (*nq_context*)

argument information

```
int PMDFabortMessage(PMDF_nq **nq_context)
```

ARGUMENTS *nq_context*
 A message enqueue context created with PMDFstartMessageEnvelope.

DESCRIPTION PMDFabortMessage aborts the specified message enqueue context, deleting the message associated with that context. The specified enqueue context is no longer usable; a new one can be generated with PMDFstartMessageEnvelope.

This routine is typically called when an error occurs while enqueueing a message and the submission needs to be aborted.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFabortProgram

PMDFabortProgram

Output an error message and then abort the currently running program.

PASCAL **PMDF_abort_program** (*message, error_code*)

argument information

Argument	Data type	Access	Mechanism
message	descriptor	read	reference
error_code	signed longword	read	value

C **PMDFabortProgram**
 (*message, message_len, error_code*)

argument information

```
void PMDFabortProgram(char *message,  
                       int  message_len,  
                       int  error_code)
```

ARGUMENTS

message

A text string to output as an error message. The length of this string should not exceed `SHORTALFA_SIZE` bytes. Any string exceeding this length will be truncated to `SHORTALFA_SIZE` bytes.

message_len

Length in bytes of **message**.

error_code

An integer error code to output as part of the error message. If **error_code** is 0, it will not be output.

DESCRIPTION

`PMDFabortProgram` outputs as an error message the supplied text string **message** and, if non-zero, **error_code**. After the error message is output, a halt instruction is issued thereby aborting the currently running program. Generally, this routine should only be called when an unrecoverable error has been detected. Before calling `PMDFabortProgram`, any active message enqueue or dequeue contexts should be aborted with `PMDFabortMessage` or `PMDFdequeueMessageEnd`. Note that this routine can be called even when `PMDFinitialize` has failed.

On OpenVMS systems, the error message is written to `PMDF_OUTPUT` if defined and `SYS$OUTPUT` otherwise. On UNIX systems, the error message is written to

stdout.

Example output generated on an OpenVMS system in response to the call
PMDFabortProgram("Fatal error in BITBUCKET channel", 8922);

is shown below:

04-MAY-2012 18:04:00: Fatal error in BITBUCKET channel, status = 8922.

%PAS-F-HALT, HALT procedure called

%TRACE-F-TRACEBACK, symbolic stack dump follows

module name	routine name	line	rel PC	abs PC
MMMOD	MM_ABORT_PROGRAM_INT	13141	00000082	00036642
PMDF_API	PMDF_ABORT_PROGRAM	5107	00000064	00009BF8
BITBUCKET	ROUND_FILE	214	00000031	00007051

RETURN VALUESNone.

PMDFaddHeaderLine

Add a header line to a header structure.

PASCAL *status* = **PMDF_add_header_line** (*header, type, line*)

argument information

Argument	Data type	Access	Mechanism
header	header pointer	write	reference
type	signed longword	read	value
line	descriptor	read	reference

C *status* = **PMDFaddHeaderLine**
 (*header, type, line, line_len*)

argument information

```
int PMDFaddHeaderLine(PMDF_hdr **header,  
                      int         type,  
                      char        *line,  
                      int         line_len)
```

ARGUMENTS *header*
Address of a header structure.

type
The type of header line being added.

line
The header line to add. No length limit is imposed.

line_len
The length in bytes of *line*.

DESCRIPTION PMDFaddHeaderLine adds a header line of the specified type to the header structure, **header**. The header structure need not have been created by a previous call to PMDFreadHeader; PMDFaddHeaderLine will initialize the structure if it is nil (zero) on input.

The type argument specifies the type of header line being added (*e.g.*, HL_FROM, HL_TO, HL_DATE, *etc.*). The accepted types are defined in the API include files; see Section 1.6 for further details. Specify HL_OTHER for a header line type not recognized by the API. Only the body of the header line must be specified in the

line argument. The field name and colon and space will be prepended to what you specify. For example, if you specify HL_X_YOW in the type argument, and the string “Wow! PMDF is great!” in the line argument, this routine will add the following header: “X-Yow: Wow! PMDF is great!”.

Header structures can be output with `PMDFwriteHeader` and disposed of with `PMDFdisposeHeader`. See Section 1.6 for further details on using and manipulating header structures.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__HEANOTKNW</code>	Unknown header line type. No header line added. Recall <code>PMDFaddHeaderLine</code> specifying <code>HL_OTHER</code> for the header line type.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for line : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. No header line added.

PMDFaddRecipient

PMDFaddRecipient

Associate a "To:", "Cc:", or "Bcc:" address with a message.

PASCAL *status = PMDF_add_recipient*
 (*nq_context, address, orig_address*)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference
address	descriptor	read	reference
orig_address	descriptor	read	reference

C *status = PMDFaddRecipient*
 (*nq_context, address, address_len, orig_address,*
 orig_address_len)

argument information

```
int PMDFaddRecipient(PMDF_nq **nq_context,  
                    char *address,  
                    int address_len,  
                    char *orig_address,  
                    int orig_address_len)
```

ARGUMENTS

nq_context

A message enqueue context created with `PMDFstartMessageEnvelope`.

address

The "To:", "Cc:", or "Bcc:" address to associate with the message. The length of the address can not exceed `ALFA_SIZE` bytes.

address_len

The length in bytes of ***address***.

orig_address

If known, the original form of the input address, ***address***. Length can not exceed `ALFA_SIZE` bytes.

orig_address_len

Length in bytes of the original address. Supply a value of zero if the original address is not known.

DESCRIPTION When enqueueing a mail message, the list of "To:", "Cc:", and "Bcc:" recipients is built up, one address at a time, by repeatedly calling `PMDFaddRecipient`. This information is then used to construct the message's envelope "To:" address list as well as the "To:", "Cc:", and "Bcc:" header lines which will appear in the message's header. Note that in the message envelope, there is no distinction between "To:", "Cc:", and "Bcc:" addresses.

The routine `PMDFsetRecipientType` is used to specify whether each address is a "To:", "Cc:", or "Bcc:" address and whether or not it should be included in the message's envelope "To:" address list. `PMDFsetRecipientType` should be called prior to `PMDFaddRecipient`; that is, `PMDFsetRecipientType` sets information for the next recipient added with `PMDFaddRecipient`. If `PMDFsetRecipientType` is never called, then each address will be treated as a "To:" address and added to the message's list of envelope "To:" addresses.

The routine `PMDFsetRecipientFlags` is used to specify NOTARY flags for a envelope "To:" address. `PMDFsetRecipientFlags` should be called prior to `PMDFaddRecipient`. If `PMDFsetRecipientFlags` is never called, then each address will be assume the NOTARY flags `PMDF_RECEIPT_FAILURES` and `PMDF_RECEIPT_DELAYS`.

After calling `PMDFstartMessageEnvelope`, `PMDFaddRecipient` should be called once for each forward pointing address ("To:", "Cc:", or "Bcc:") to be specified. Each address should conform to RFC 822. PMDF will do its best to transform non-conformant addresses into legal RFC 822 addresses; however, this is not always possible and a `PMDF__HOST` or `PMDF__PARSE` error can result. After all addresses have been specified, then `PMDFstartMessageHeader` should be called, after which no more addresses can be specified for the current message.

While multiple addresses, separated by commas, can be passed in a single call, specifying one address per call is recommended: when multiple addresses are specified and an error results, it is not possible to determine which address was in error.

Note also that the same address can be specified more than once. This can or can not result in multiple copies of the message being sent to that address. PMDF itself will attempt to deliver a copy of the message to each instance of a specified address; however, some mail systems receiving the mail can only deliver a single copy of the message to each recipient, regardless of how many times a recipient appears in the envelope "To:" address list (e.g., VMS MAIL).

RETURN VALUES

<code>PMDF__OK</code>	Successful, normal completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. No address added.
<code>PMDF__HOST</code>	Illegal address. No address added. Call <code>PMDFgetErrorText</code> to obtain specific information about the nature of the error.

PMDfAddRecipient

PMDf__INVSTRDES	Invalid string descriptor for address : descriptor has an invalid value in its DSC\$B_CLASS field. No address added.
PMDf__NAUTH	Sender is not authorized to send to this address or mailing list. No address added. Call <code>PMDfgetErrorText</code> to obtain specific information about the nature of the error.
PMDf__PARSE	Bad address syntax. No address added. Call <code>PMDfgetErrorText</code> to obtain specific information about the nature of the error.
PMDf__STRTRUERR	Length of address exceeds ALFA_SIZE bytes. No address added.
PMDf__USER	Unknown or illegal user name specification. No address added. Call <code>PMDfgetErrorText</code> to obtain specific information about the nature of the error.

PMDFAAddressDispose

Dispose of an address context.

PASCAL *status* = **PMDF_address_dispose** (*addr_context*)

argument information

Argument	Data type	Access	Mechanism
<i>addr_context</i>	context pointer	read	value

C *status* = **PMDFAAddressDispose** (*addr_context*)

argument information

```
int PMDFAAddressDispose(PMDF_addr *addr_context)
```

ARGUMENTS *addr_context*
 Address context generated by a previous call to PMDFAAddressParseList.

DESCRIPTION Address contexts created with PMDFAAddressParseList must be disposed of by calling PMDFAAddressDispose. Failure to do so will result in memory leaks.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFaddressGet

PMDFaddressGet

Extract an address from a list of parsed addresses.

PASCAL *status* = **PMDF_address_get**
 (*addr_context*, *index*, *address*, *address_len*)

argument information

Argument	Data type	Access	Mechanism
<i>addr_context</i>	context pointer	read	value
<i>index</i>	integer	read	value
<i>address</i>	descriptor	read/write	reference
<i>address_len</i>	unsigned word	write	reference

C *status* = **PMDFaddressGet**
 (*addr_context*, *index*, *address*, *address_len*)

argument information

```
int PMDFaddressGet(PMDF_addr *addr_context,  
                  int index,  
                  char *address,  
                  int *address_len)
```

ARGUMENTS ***addr_context***
Address context generated by a previous call to `PMDFaddressParseList`.

index
Index of the address to extract from the list of parsed addresses.

address
String to receive the extracted address. Must be at least `ALFA_SIZE` bytes in length for *PMDF_address_get* and `ALFA_SIZE+1` bytes for `PMDFaddressGet`.

line_len
Length in bytes of the returned address. Callers using `PMDFaddressGet` must, on input, supply the maximum length in bytes of ***address***.

DESCRIPTION After parsing a line of addresses with `PMDFAccessParseList`, the individual addresses can each be retrieved with `PMDFAccessGet`. Call `PMDFAccessGet` once for each address. The `index` argument can range from 1 to **count** where **count** is the count of parsed addresses returned by `PMDFAccessParseList`. The first address corresponds to an **index** value of 1 and the last to an **index** value of **count**.

Note that `PMDFAccessGet` will also heuristically correct addresses with minor syntactical problems.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__NO</code>	Value for index is out of range. No address returned.

PMDFAddressGetProperty

PMDFAddressGetProperty

Extract a property of an address from a list of parsed addresses.

PASCAL *status* = **PMDF_address_get_property**
 (*addr_context*, *index*, *property*, *result*, *result_len*)

argument information

Argument	Data type	Access	Mechanism
<i>addr_context</i>	context pointer	read	value
<i>index</i>	integer	read	value
<i>property</i>	integer	read	value
<i>result</i>	descriptor	read/write	reference
<i>result_len</i>	unsigned word	write	reference

C *status* = **PMDFAddressGetProperty**
 (*addr_context*, *index*, *property*, *result*, *result_len*)

argument information

```
int PMDFAddressGetProperty(PMDF_addr *addr_context,  
                           int index,  
                           int property,  
                           char *result,  
                           int *result_len)
```

ARGUMENTS

addr_context

Address context generated by a previous call to `PMDFAddressParseList`.

index

Index of the address to obtain the property for.

property

The address property to return.

result

String to receive the address property. Must be at least `ALFA_SIZE` bytes in length for `PMDF_address_get_property` and `ALFA_SIZE+1` bytes for `PMDFAddressGetProperty`.

result_len

Length in bytes of the returned property. Callers using `PMDFAddressGetProperty` must, on input, supply the maximum length in bytes of ***result***.

DESCRIPTION After parsing a line of addresses with `PMDFaddressParseList`, properties of individual addresses can be retrieved with `PMDFaddressGetProperty`. The `index` argument can range from 1 to **count** where **count** is the count of parsed addresses returned by `PMDFaddressParseList`. The first address corresponds to an **index** value of 1.

The accepted values for **property** are listed and described in the table below. Note that unlike `PMDFgetAddressProperty`, `PMDFaddressGetProperty` does not accept the `PMDF_PROP_FRIENDLY` property.

Table 1–3 Properties of the Address *phrase* <*@otherhost:user@host*>

Symbolic name	Value	Description
<code>PMDF_PROP_ADDRESS</code>	1	Address part, <i>@otherhost:user@host</i> , of the address
<code>PMDF_PROP_DOMAIN</code>	2	Domain part, <i>host</i> , of the address
<code>PMDF_PROP_LOCAL</code>	4	Local part, <i>user</i> , of the address
<code>PMDF_PROP_PHRASE</code>	5	Phrase part, <i>phrase</i> , of the address, if any
<code>PMDF_PROP_PROPER</code>	6	Full address including any phrases and comments
<code>PMDF_PROP_ROUTE</code>	7	Source route part, <i>@otherhost:</i> , of the address, if any

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BAD</code>	Bad parameter supplied: invalid value for property . No result returned.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No result returned.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. No result returned.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for result : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. No result returned.
<code>PMDF__NO</code>	Value for index is out of range. No result returned.
<code>PMDF__STRTRU</code>	Supplied string was too long; result truncated to fit.

PMDFAddressParseList

Parse a line of comma separated addresses.

PASCAL *status* = **PMDF_address_parse_list**
 (*addr_context*, *count*, *line*)

argument information

Argument	Data type	Access	Mechanism
<i>addr_context</i>	context pointer	read/write	reference
<i>count</i>	integer	write	reference
<i>line</i>	descriptor	read	reference

C *status* = **PMDFAddressParseList**
 (*addr_context*, *count*, *line*, *line_len*)

argument information

```
int PMDFAddressParseList(PMDF_addr **addr_context,  
                          int *count,  
                          char *line,  
                          int line_len)
```

ARGUMENTS *addr_context*
Address context created for the parsed address line.

count
The number of addresses parsed.

line
Character string containing the list of comma separated, RFC 822 addresses to be parsed.

line_len
Length in bytes of the string of addresses to parse.

DESCRIPTION **PMDFAddressParseList** can be used to parse a line of one or more comma separated RFC 822 addresses. The input line can be of arbitrary length. The result of the parse is represented by an address context, **addr_context**, and a count of parsed addresses, **count**. Each parsed address can then be individually extracted from the parsed line with a call to **PMDFAddressGet** or

PMDFaddressGetProperty. The address context should be disposed of with a call to PMDFaddressDispose.

When there are no valid addresses in the input line, the returned context will be zero (nil) and the count zero.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__INVSTRDES	Invalid string descriptor for line : descriptor has an invalid value in its DSC\$B_CLASS field. No result returned.

PMDFaliasNoExpansion

PMDFaliasNoExpansion

Inhibit the expansion of aliases for all subsequent recipient addresses.

PASCAL *status* = **PMDF_alias_no_expansion** (*nq_context*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference

C *status* = **PMDFaliasNoExpansion** (*nq_context*)

argument information

```
int PMDFaliasNoExpansion(PMDF_nq **nq_context)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

DESCRIPTION Generally, expansion of aliases should never be inhibited. However, there is at least one situation where alias expansion must be inhibited. That situation arises when a message being dequeued needs to be re-enqueued to the same channel with a subset of its envelope recipient list. That is done by enqueueing a new message, inhibiting alias expansion, specifying only the envelope addresses of the desired subset, ending the envelope, and then copying the message content verbatim with `PMDFcopyMessage`.

RETURN VALUES

`PMDF__OK` Normal, successful completion.

PMDFcancelCallback

Cancel the use of any specified call back routines.

PASCAL *status* = **PMDF_cancel_call_back**

C *status* = **PMDFcancelCallback** ()

argument information

```
int PMDFcancelCallback()
```

ARGUMENTS *None.*

DESCRIPTION After calling `PMDFcancelCallback`, the API will no longer invoke any call back routines specified with a previous call to `PMDFsetCallback`.

On UNIX systems, this routine merely returns `PMDF__OK`.

RETURN VALUES

`PMDF__OK` Normal, successful completion.

PMDFcloseLogFile

PMDFcloseLogFile

Close the PMDF log file if it is open.

PASCAL *status* = **PMDF_close_log_file**

C *status* = **PMDFcloseLogFile** ()

argument information

```
int PMDFcloseLogFile()
```

ARGUMENTS *None.*

DESCRIPTION PMDFcloseLogFile can be called to close the PMDF log file if it is open. Only programs which (1) enqueue or dequeue mail and (2) run indefinitely before ever calling PMDFdone, need worry about calling PMDFcloseLogFile. See Section 1.7 for a discussion of this topic.

Note that the PMDF log file is distinct from channel log files. The PMDFlog routine is not related to the PMDFcloseLogFile routine.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFcloseQueueCache

Close the queue cache database if it is open.

PASCAL *status* = **PMDF_close_queue_cache**

C *status* = **PMDFcloseQueueCache** ()

argument information

```
int PMDFcloseQueueCache()
```

ARGUMENTS *None.*

DESCRIPTION *PMDFcloseQueueCache* can be called to close the *PMDF* queue cache database if it is open. Only programs which (1) enqueue or dequeue mail and (2) run indefinitely before ever calling *PMDFdone*, need worry about calling *PMDFcloseQueueCache*. See Section 1.7 for a discussion of this topic.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFcopyMessage

Make a verbatim copy of a message header and content.

PASCAL *status* = **PMDF_copy_message**
 (*dq_context*, *nq_context*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>nq_context</i>	context pointer	read/write	reference

C *status* = **PMDFcopyMessage**
 (*dq_context*, *nq_context*)

argument information

```
int PMDFcopyMessage(PMDF_dq **dq_context,  
                    PMDF_nq **nq_context)
```

ARGUMENTS *dq_context*
 A message dequeue context created with `PMDFdequeueInitialize`.

nq_context
 A message enqueue context created with `PMDFstartMessageEnvelope`.

DESCRIPTION Use `PMDFcopyMessage` to efficiently copy to a new message being enqueued a verbatim copy of a message being dequeued. Only the portion of the dequeued message following the read point for that message will be copied. Thus, if the entire dequeued message — header and content — is to be copied, then it can be necessary to first call `PMDFrewindMessage`.

`PMDFcopyMessage` is especially useful in cases where a message needs to have its envelope changed but be left enqueued. For example, when a message was successfully delivered to some but not all recipients. In that case, if some of the recipients could not be delivered to owing to temporary problems, the message should be re-enqueued verbatim to just those recipients who could not be handled because of temporary problems. In such a case, be sure to also call `PMDFaliasNoExpansion` while enqueueing the new message.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Bad value passed for dq_context or nq_context .

PMDFdatabaseAddEntry

PMDFdatabaseAddEntry

Add an entry to a database.

PASCAL *status = PMDF_database_add_entry*
 (*database, entry, value, create_db, replace, setbits, bits*)

argument information

Argument	Data type	Access	Mechanism
database	signed longword	read	value
entry	descriptor	read	reference
value	descriptor	read	reference
create_db	boolean	read	value
replace	boolean	read	value
setbits	boolean	read	value
bits	unsigned longword	read	value

C *status = PMDFdatabaseAddEntry*
 (*database, entry, entry_len, value, value_len, create_db,*
 replace, setbits, bits)

argument information

```
int PMDFdatabaseAddEntry(int      database,  
                          char     *entry,  
                          int      entry_len,  
                          char     *value,  
                          int      value_len,  
                          int      create_db,  
                          int      replace,  
                          int      setbits,  
                          unsigned long int bits)
```

ARGUMENTS *database*
Database to add the entry to.

entry
Entry to add to the database (*e.g.*, alias name). Length of this string should not exceed `KEY_LENGTH` for a short database or `LONG_KEY_LENGTH` for a long database.

entry_len
Length in bytes of the entry.

value

Value to associate with the database entry (e.g., alias translation value). Length of this string should not exceed DATA_LENGTH for a short database or LONG_DATA_LENGTH for a long database.

value_len

Length in bytes of the entry's value.

create_db

When true, create the database if it does not already exist.

replace

When true, replace the entry if one already exists in the database.

setbits

When true, set control bits associated with a personal alias database entry.

bits

Integer longword containing personal alias control bits.

DESCRIPTION

PMDFdatabaseAddEntry adds an entry to a database. If the database is not already opened, it will be opened. When no more database accesses are to be performed, the database should be closed with PMDFdatabaseClose.

The specified entry and its associated value will be added to the database. If the database does not exist, then it will be created if **create_db** is true; otherwise, a PMDF__CANOPNDAT error will be returned and no database created. When a database is created, it will be created as a long database, if possible, and as a short database if not. If the specified entry already exists in the database, then it will be replaced if **replace** is true; otherwise, a PMDF__CANTUPDAT error will be returned and no entry added.

The length of the entry and its value can not exceed, respectively, the key and data lengths used by the database. PMDF databases come in two sizes: short and long. A short database uses a key length of KEY_LENGTH and a data length of DATA_LENGTH. A long database uses a key length of LONG_KEY_LENGTH and a data length of LONG_DATA_LENGTH. The values of these constants are given in Table 1-2.

The database to use is specified with the **database** argument. The possible values for that argument are shown in Table 1-4. In that table, the second column gives the symbolic names for the different databases, as defined in the API include files described in Section 1.11. Whenever possible, programmers should use the symbolic names rather than the actual values.

Table 1-4 Database Symbolic Names and Values

Database	Symbolic name	Value
Alias	PMDF_DATABASE_ALIAS	1
Domain	PMDF_DATABASE_DOMAIN	3
PMDF-MR FROM_MR	PMDF_DATABASE_FROM_MR	4

PMDFdatabaseAddEntry

Table 1–4 (Cont.) Database Symbolic Names and Values

Database	Symbolic name	Value
PMDF-X400 FROM_X400	PMDF_DATABASE_FROM_X400	5
General	PMDF_DATABASE_GENERAL	6
Personal alias	PMDF_DATABASE_PERSONAL_ALIAS	7
Address reversal	PMDF_DATABASE_REVERSE	8
PMDF-MR TO_MR	PMDF_DATABASE_TO_MR	9
PMDF-X400 TO_X400	PMDF_DATABASE_TO_X400	11
User profile	PMDF_DATABASE_USER_PROFILE	12
Popstore forward	PMDF_DATABASE_POPSTORE_FORWARD	13
Pipe	PMDF_DATABASE_PIPE	15
Forward	PMDF_DATABASE_FORWARD	16

The **setbits** and **bits** arguments are for use only with personal alias databases. When **setbits** is true, the control bits specified in the bit mask **bits** will be set for the alias. In this case, the length of **value** can not exceed `DATA_LENGTH - 4` or `LONG_DATA_LENGTH - 4`. The bits in the bit mask **bits** control aspects of the alias and are shown in the table below:

Bit	Usage
PMDF_ALIAS_ADDRESS_BIT	Alias translation value is an address or mailing list
PMDF_ALIAS_FAX_BIT	Alias translation value is a FAX address
PMDF_ALIAS_PUBLIC_BIT	When set, alias is public; when clear, alias is private
PMDF_ALIAS_EXPAND_BIT	When set, alias is expanded in message headers; when clear, alias is not expanded
PMDF_ALIAS_RECEIPT_BIT	When set, receipts are allowed to pass through; when clear, receipts are blocked

PMDF_ALIAS_ADDRESS_BIT should always be set; otherwise, it will not be possible to manipulate the resulting alias from within the PMDF DB utility. PMDF_ALIAS_FAX_BIT should be set if the alias is to be manipulated from within PMDF DB's FAX mode. The API include files described in Section 1.11 provide values for the PMDF_ALIAS_ symbolic names.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	Bad parameter value: illegal value specified for database .
PMDF__CANOPNDAT	Database could not be opened. If it does not exist, then create_db must be true in order to force the creation of the database.
PMDF__CANTUPDAT	Cannot update the database. Attempt to add or replace the entry failed.
PMDF__DUPENTRY	Entry already exists and replace was false. No entry added.
PMDF__ENTWONFIT	Length of entry or value too long for database. No entry added.

PMDFdatabaseAddEntry

PMDF__INVSTRDES

Invalid string descriptor for **entry** or **value**: one or both descriptors has an invalid value in its DSC\$B_CLASS field. No entry added.

PMDF__STRTRUERR

Supplied string **entry** or **value** is too long. No entry added.

PMDFdatabaseClose

PMDFdatabaseClose

Close a database.

PASCAL *status* = **PMDF_database_close** (*database*)

argument information

Argument	Data type	Access	Mechanism
database	signed longword	read	value

C *status* = **PMDFdatabaseClose** (*database*)

argument information

```
int PMDFdatabaseClose(int database)
```

ARGUMENTS *database*
Database to close.

DESCRIPTION PMDFdatabaseClose should be called to close a database opened with a PMDF-databaseAddEntry, PMDFdatabaseDeleteEntry, or PMDFdatabaseGetEntry call.

See the description of PMDFdatabaseAddEntry for a list of the legal values for **database**.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	Bad parameter value: illegal value specified for database .

PMDFdatabaseDeleteEntry

Remove an entry from a database.

PASCAL *status* = **PMDF_database_delete_entry**
 (*database*, *entry*)

argument information

Argument	Data type	Access	Mechanism
database	signed longword	read	value
entry	descriptor	read	reference

C *status* = **PMDFdatabaseDeleteEntry**
 (*database*, *entry*, *entry_len*)

argument information

```
int PMDFdatabaseDeleteEntry(int  database,
                             char *entry,
                             int  entry_len)
```

ARGUMENTS

database

Database to delete the entry from.

entry

Entry to remove from the database (*e.g.*, an alias). Length of this string should not exceed `KEY_LENGTH` for a short database or `LONG_KEY_LENGTH` for a long database.

entry_len

Length in bytes of the entry.

DESCRIPTION

Entries are removed from databases with `PMDFdatabaseDeleteEntry`. In the case of duplicate entries, multiple calls are required to remove all entries — one call per entry. If the specified database is not already opened, then it will be opened automatically. When no more database accesses are to be performed, the database should be closed with `PMDFdatabaseClose`.

See the description of `PMDFdatabaseAddEntry` for a list of the legal values for **database**.

PMDFdatabaseDeleteEntry

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	Bad parameter value: illegal value specified for database . No entry deleted.
PMDF__NO	No matching entry found. No entry deleted.
PMDF__CANOPNDAT	Database could not be opened or does not exist.
PMDF__CANTUPDAT	Cannot update the database. Attempt to delete an entry failed.
PMDF__ENTWONFIT	Length of entry too long for database. No entry deleted.
PMDF__INVSTRDES	Invalid string descriptor for entry : descriptor has an invalid value in its DSC\$B_CLASS field. No entry deleted.
PMDF__STRTRUERR	Supplied string entry is too long. No entry deleted.

PMDFdatabaseGetEntry

Lookup an entry in a database.

PASCAL *status* = **PMDF_database_get_entry**
 (*database, access, entry, entry_len, value, value_len, bits*)

argument information

Argument	Data type	Access	Mechanism
database	signed longword	read	value
access	signed longword	read	value
entry	descriptor	read/write	reference
entry_len	unsigned word	read/write	reference
value	descriptor	write	reference
value_len	unsigned word	write	reference
bits	unsigned longword	write	reference

C *status* = **PMDFdatabaseGetEntry**
 (*database, access, entry, entry_len, value, value_len, bits*)

argument information

```
int PMDFdatabaseGetEntry(int      database,
                          int      access,
                          char     *entry,
                          int      *entry_len,
                          char     *value,
                          int      *value_len,
                          unsigned long int *bits)
```

ARGUMENTS

database
 Database to search.

access
 Type of search to perform.

entry
 Entry to search for in the database. Length of this should be `KEY_LENGTH` for a short database or `LONG_KEY_LENGTH` for a long database. On output the actual entry read from the database will be returned in **entry**.

entry_len
 On input, the length in bytes of the entry. On output, the length in bytes of the returned entry.

PMDFdatabaseGetEntry

value

Value of the entry retrieved from the database. Length must be at least `LONG_DATA_LENGTH` bytes for `PMDF_database_get_entry` or `LONG_DATA_LENGTH+1` bytes for `PMDFdatabaseGetEntry`.)

value_len

Length in bytes of the returned entry value. Callers using `PMDFdatabaseGetEntry` must, on input, supply the maximum length in bytes of **value**.

bits

Optional integer longword containing personal alias control bits associated with the returned value.

DESCRIPTION

`PMDFdatabaseGetEntry` can be called to find an entry in a database and return the value associated with the entry. If the database is not already opened, it will be opened. When no more database accesses are to be performed, it should be closed with `PMDFdatabaseClose`.

The first time a given entry is to be located, **access** should have the value `PMDF_DATABASE_GET_FIRST` or `PMDF_DATABASE_GET_FIRST_ROOT`. If a matching entry is found, then the return status code will be `PMDF__OK`. If no match is found or the database could not be opened (*e.g.*, does not exist), then `PMDF__EOF` will be returned. To search for any additional matching entries, make repeated calls specifying either `PMDF_DATABASE_GET_NEXT` or `PMDF_DATABASE_GET_NEXT_ROOT` for **access**. After no more matching entries can be found, a status code of `PMDF__EOF` will be returned.

The **access** argument specifies the nature of the database search to perform. The possible values for **access** are shown in the table below. In that table, the second column gives the symbolic names for the different access types. These symbolic names are defined in the API include files described in Section 1.11. Whenever possible, programmers should use the symbolic names rather than the actual values.

Symbolic name	Value	Description
<code>PMDF_DATABASE_GET_FIRST</code>	1	Starting from the beginning of the database, find the first database entry which matches (case blind) entry .
<code>PMDF_DATABASE_GET_NEXT</code>	2	Continuing from the last located entry, find the next database entry which matches (case blind) entry .
<code>PMDF_DATABASE_GET_FIRST_ROOT</code>	3	Starting from the beginning of the database, find the first database entry whose first entry_len characters match (case blind) entry .
<code>PMDF_DATABASE_GET_NEXT_ROOT</code>	4	Continuing from the last located entry, find the next database entry whose first entry_len characters match (case blind) entry .
<code>PMDF_DATABASE_GET_FIRST_ALL</code>	5	Return the first entry from the database.
<code>PMDF_DATABASE_GET_NEXT_ALL</code>	6	Return the next entry from the database.

The **bits** argument is optional and only used in conjunction with personal alias databases. When an alias value is returned, any control bits associated with

that alias will be returned in the bit mask **bits**. Consult the description of `PMDFdatabaseAddEntry` for details on this bit mask.

To retrieve all entries from a database use `PMDF_DATABASE_GET_FIRST_ALL` and `PMDF_DATABASE_GET_NEXT_ALL`.

See the description of `PMDFdatabaseAddEntry` for a list of the legal values for **database**.

Note: For each PMDF database, a single per-process read context is maintained by PMDF. As such, any sequence of chained `PMDFdatabaseGetEntry` calls must not be interrupted by other threads accessing the same database with `PMDFdatabase` calls. Any interruption will disrupt the read state. A chained sequence is one that starts with a `PMDF_DATABASE_GET_FIRST` or `PMDF_DATABASE_GET_FIRST_ROOT` access followed by either a `PMDF_DATABASE_GET_NEXT` or `PMDF_DATABASE_GET_NEXT_ROOT` access to find subsequent, related entries.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BAD</code>	Bad parameter value: illegal value specified for database or access . No database search performed; no value returned.
<code>PMDF__EOF</code>	No matching entry found; no value returned.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No value returned.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. No value returned.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for entry or value : one or both descriptors has an invalid value in its <code>DSC\$B_CLASS</code> field. No value returned.
<code>PMDF__STRTRU</code>	Supplied string value is too short; output truncated to fit into string.
<code>PMDF__STRTRUERR</code>	Supplied string entry is too long. No value returned.

PMDFdebug

Enable debugging output.

PASCAL *status = PMDF_debug*
 (*enqueue_debug, dequeue_debug*)

argument information

Argument	Data type	Access	Mechanism
enqueue_debug	boolean	read	value
dequeue_debug	boolean	read	value

C *status = PMDFdebug*
 (*enqueue_debug, dequeue_debug*)

argument information

```
int PMDFdebug(int enqueue_debug, int dequeue_debug)
```

ARGUMENTS *enqueue_debug*
When true, enables message enqueue debugging output. When false, disables message enqueue debugging output.

dequeue_debug
When true, enables message dequeue debugging output. When false, disables message dequeue debugging output.

DESCRIPTION PMDF is capable of producing voluminous debugging output both while enqueueing and dequeuing messages. By default, this output is disabled. To enable either enqueue or dequeue debugging output, call `PMDFdebug` with the appropriate argument set true.

Since any of the routines `PMDFinitialize`, `PMDFenqueueInitialize`, or `PMDFdequeueInitialize` can explicitly initialize the debugging flags, `PMDFdebug` should be called after calls to those routines have been made.

Note that output of additional debugging information can be enabled by setting `OS_DEBUG=1` in the PMDF option file. Setting `DEQUEUE_DEBUG=1` in the PMDF option file is equivalent to setting the dequeuing debug flag to true with

PMDFdebug; a similar relation holds between the PMDF option `MM_DEBUG` and the enqueueing debug flag.

On OpenVMS systems, the debugging output will be written to `PMDF_DEBUG`: if defined and `SYS$OUTPUT` otherwise. On UNIX systems, debugging output will be written to `stdout`.

RETURN VALUES

`PMDF__OK`

Normal, successful completion.

PMDFdecodeMessage

Decode a MIME formatted message.

PASCAL *status* = **PMDF_decode_message**
 (*dq_context*, *param*, *flags*, *input_line*, *output_header*,
 output_line, *output_block*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>param</i>	address pointer	read	value
<i>flags</i>	unsigned longword	read	value
<i>input_line</i>	procedure	read	reference
<i>output_header</i>	procedure	read	reference
<i>output_line</i>	procedure	read	reference
<i>output_block</i>	procedure	read	reference

C *status* = **PMDFdecodeMessage**
 (*dq_context*, *param*, *flags*, *input_line*, *output_header*,
 output_line, *output_block*)

argument information

```
int PMDFdecodeMessage(PMDF_dq        **dq_context,  
                      void            *param,  
                      unsigned long  flags,  
                      void            (*input_line)(),  
                      void            (*output_header)(),  
                      void            (*output_line)(),  
                      void            (*output_block)())
```

ARGUMENTS *dq_context*
Optional message dequeue context created with `PMDFdequeueInitialize`. If not specified, then ***input_line*** must be specified.

param
Optional parameter which will be passed to each of the supplied routines, ***input_line***, ***output_header***, ***output_line***, and ***output_block***, when they are called.

flags
Bit flags controlling the operation of `PMDFdecodeMessage`.

input_line

Optional address of a procedure to read each line of the message to be decoded. If not specified, then **dq_context** must be specified.

output_header

Address of a procedure to output either the outer message header or the header associated with a message part.

output_line

Address of a procedure to output a line of the content of a non-binary message part.

output_block

Address of a procedure to output a block of data from a binary message part.

DESCRIPTION

PMDFdecodeMessage can be used to decode a MIME message. Example programs illustrating the use of this routine are given in the files `api_example9.pas` and `api_example10.c` and can be found in the `PMDF_ROOT:[DOC.EXAMPLES]` directory on OpenVMS systems or, on UNIX systems, in the `/pmdf/doc/examples` directory.

Each line of the message to be decoded can come from either a message currently being dequeued or from an arbitrary source. If the former, then supply the message dequeue context generated by `PMDFdequeueInitialize` and specify zero for the **input_line** argument. The message being dequeued must have its read point positioned at the start of the message's outer header. That is the position the read point will be at after the last envelope recipient address has been read with `PMDFgetRecipient` or after calling `PMDFrewindMessage`.

To decode a message from an arbitrary source, specify zero for the **dq_context** argument, and supply with **input_line** the address of a procedure to call to obtain each successive line of the message. The input procedure must be of the form

```
int input_line(void *param, char *line, int *line_len)
```

When the procedure is called, `param` will have the value of the parameter supplied to `PMDFdecodeMessage` with the **param** argument, `line` will be the address of a buffer to place the message line into, and `*line_len` will be the maximum number of bytes which can be written to the buffer. On output, the procedure should return in `*line_len` the number of bytes placed into the buffer. The buffer does not need to be zero terminated. Finally the procedure should return a value of `PMDF__OK` if there is more data to read and `PMDF__EOF` if there is an error or no further data to read.

The procedures referenced by **output_header**, **output_line**, and **output_block** have the form

```
int output_header(void *param,
                  PMDF_hdr *hdr,
                  int part,
                  int depth,
                  int index)
```

PMDFdecodeMessage

```
int output_line(void *param,
                char *line,
                int  line_len,
                int  eol)

int output_block(void *param,
                 char *data,
                 int  data_len,
                 int  eol)
```

where the arguments are as follows:

param	Value passed to PMDF_decode_message for the param argument.
hdr	Pointer to a PMDF_hdr structure containing the header lines to output.
part	Will have the value 2 if the message part associated with the header is textual in nature and the value 1 if the associated part is binary in nature.
depth	Nesting depth in the MIME structure for this message part.
index	Index for this part; first message part at depth N has an index value of 1, second part at depth N has an index value of 2, etc..
line	Line of text output. This text comes from the content of a non-binary message part. The line is not null terminated.
line_len	Length in bytes of the line of message text to output.
eol	Binary. Indicates end-of-line was seen.
data	Raw binary data to output. This data comes from the content of a binary data part. The data is not null terminated and can contain nulls within it.
data_len	Length in bytes of the data to output.

The output routines should return an odd-valued result (*e.g.*, 1, PMDF__OK) when successful, and an even-valued result otherwise (*e.g.*, 0, PMDF__NO). When an even-valued result is returned by an output routine, PMDFdecodeMessage will abort the decode operation and return to the caller the value returned by the output routine.

When the lowest bit of **flags** is set to 1, a message in any of the various formats which PMDF understands (*e.g.*, RFC 1154, Pathworks, NeXT, etc.) will be first translated to MIME prior to decoding. Furthermore, if the message does not have a recognized format, but does contain embedded information encoded with UUENCODE or BINHEX, then the message will be converted to MIME prior to decoding with the encoded material placed in a separate attachment.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	Both dq_context and input_line were zero. Message not decoded.
PMDF__BADCONTEXT	Invalid dq_context supplied. Message not decoded.

PMDFdeferMessage

Defer a message for later processing.

PASCAL *status* = **PMDF_defer_message**
 (*dq_context*, *increment*, *reason*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>increment</i>	boolean	read	value
<i>reason</i>	descriptor	read	reference

C *status* = **PMDFdeferMessage**
 (*dq_context*, *increment*, *reason*, *reason_len*)

argument information

```
int PMDFdeferMessage(PMDF_dq **dq_context,
                    int      increment,
                    char     *reason,
                    int      reason_len)
```

ARGUMENTS *dq_context*
 A message dequeue context created with `PMDFdequeueInitialize`.

increment
 If true, the message's retry count will be incremented; otherwise, the retry count will be left unchanged.

reason
 Optional text string describing why the message is being deferred. The length of this string should not exceed `BIGALFA_SIZE` bytes.

reason_len
 Length in bytes of **reason**.

DESCRIPTION **NOTE:** Although still supported, this routine is now obsolete. Use the `PMDFdequeueMessageEnd` routine instead.

`PMDFdeferMessage` can be called to defer processing of the currently accessed message. The deferred message will be left in PMDF's message queues for processing by a subsequent processing job. If the message continues to remain

PMDFdeferMessage

in the message queues long enough, it will be returned by PMDF's message return system. See the message return and bouncing discussions in the *PMDF System Manager's Guide* for further details on this subject. When a message is deferred, no notification messages will be generated despite any prior calls to `PMDFrecipientDisposition`. This is because deferring a message with `PMDFdeferMessage` causes all of the message's recipient addresses to be deferred for later reprocessing.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Message not deferred.

PMDFdeleteHeaderLine

Remove all header lines of a given type from a header structure.

PASCAL *status* = **PMDF_delete_header_line** (*header*, *type*)

argument information

Argument	Data type	Access	Mechanism
header	header pointer	read	value
type	signed longword	read	value

C *status* = **PMDFdeleteHeaderLine** (*header*, *type*)

argument information

```
int PMDFdeleteHeaderLine(PMDF_hdr *header, int type)
```

ARGUMENTS

header

Address of a header structure previously created by `PMDFreadHeader` or `PMDFaddHeaderLine`.

type

The type of header line being removed.

DESCRIPTION

`PMDFdeleteHeaderLine` removes all header lines of the type **type** from the header structure pointed at by **header**. That is, the linked list

```
header[type]
```

will be disposed of.

The type argument specifies the type of header lines to be removed (*e.g.*, `HL_FROM`, `HL_TO`, `HL_DATE`, *etc.*). The accepted types are defined in the API include files; see Section 1.6 for further details. Specify `HL_OTHER` for a header line type not recognized by the API.

RETURN VALUES

`PMDF__OK`

Normal, successful completion.

PMDfdeleteHeaderLine

PMDf__HEANOTKNW

Unknown header line type. No header lines removed. Recall PMDfdeleteHeaderLine specifying HL_OTHER for the header line type.

PMDFdequeueEnd

Terminate and dispose of a PMDF dequeue context.

PASCAL *status* = **PMDF_dequeue_end** (*dq_context*)

argument information

Argument	Data type	Access	Mechanism
dq_context	context pointer	write	reference

C *status* = **PMDFdequeueEnd** (*dq_context*)

argument information

```
int PMDFdequeueEnd(PMDF_dq **dq_context)
```

ARGUMENTS *dq_context*
 Message dequeue context created for this message dequeuing context.

DESCRIPTION PMDFdequeueEnd should be called to dispose of a message dequeue context created with PMDFdequeueInitialize. This routine should be called prior to PMDFdone.

RETURN VALUES
 PMDF__OK Normal, successful completion.

PMDFdequeueInitialize

PMDFdequeueInitialize

Initialize PMDF for message dequeuing operations and create a message dequeue context.

PASCAL *status* = **PMDF_dequeue_initialize** (*dq_context*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	write	reference

C *status* = **PMDFdequeueInitialize** (*dq_context*)

argument information

```
int PMDFdequeueInitialize(PMDF_dq **dq_context)
```

ARGUMENTS *dq_context*
Message dequeue context created for this message dequeuing context.

DESCRIPTION Initialize PMDF for message dequeue operations and create a message dequeue context. `PMDFinitialize` must be called prior to calling `PMDFdequeueInitialize`; after calling `PMDFdequeueInitialize`, `PMDFgetMessage` can be called. Use `PMDFdequeueEnd` to dispose of a message dequeue context.

RETURN VALUES

`PMDF__OK` Normal, successful completion.

PMDFdequeueMessage

Remove a message from PMDF's message queues.

PASCAL *status* = **PMDF_dequeue_message** (*dq_context*)

argument information

Argument	Data type	Access	Mechanism
dq_context	context pointer	read/write	reference

C *status* = **PMDFdequeueMessage** (*dq_context*)

argument information

```
int PMDFdequeueMessage(PMDF_dq **dq_context)
```

ARGUMENTS *dq_context*
 A message dequeue context created with PMDFdequeueInitialize.

DESCRIPTION **NOTE:** Although still supported, this routine is now considered obsolete. Instead use the PMDFdequeueMessageEnd routine.

If calls to PMDFrecipientDisposition were made prior to calling PMDFdequeueMessage, then PMDF will automatically generate any required notification messages when PMDFdequeueMessage is called. Once any notification messages have been generated, the message being dequeued is then permanently removed from the message queue.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. Message not dequeued.

PMDFdequeueMessageEnd

Remove a message from PMDF's message queues.

PASCAL *status* = **PMDF_dequeue_message_end**
 (*dq_context*, *defer*, *reason*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>defer</i>	boolean	read	value
<i>reason</i>	descriptor	read	reference

C *status* = **PMDFdequeueMessageEnd**
 (*dq_context*, *defer*, *reason*, *reason_len*)

argument information

```
int PMDFdequeueMessageEnd(PMDF_dq **dq_context,  
                           int      defer,  
                           char     *reason,  
                           int      reason_len)
```

ARGUMENTS *dq_context*
A message dequeue context created with `PMDFdequeueInitialize`.

defer
When true (1), the message will be deferred for later processing.

reason
Optional text string describing why the message is being deferred. The length of this string should not exceed `BIGALFA_SIZE` bytes.

reason_len
Length in bytes of *reason*.

DESCRIPTION **NOTE:** Use of this routine with *defer* set to false (0) requires that `PMDFrecipientDisposition` be called for each recipient address obtained with `PMDFgetRecipient`.

To finish processing a message, call `PMDFdequeueMessageEnd`. This will re-enqueue the message if it requires deferred processing of some or all of its

recipients as well as generate any required notification messages concerning the message. Specifically, if all recipient addresses have a permanent disposition (PMDF_DISP_DELIVERED, _FAILED, _RELAYED, _RELAYED_FOREIGN, or _RETURN) then any required notifications are generated and the message is permanently removed from the processing queue. If all recipients are to be deferred (PMDF_DISP_DEFERRED), then no notifications are generated and the message is left in the queue for later re-processing. If some recipients have a permanent disposition while others were deferred, then

1. Any required notifications are generated for those recipients with permanent dispositions,
2. A new message is enqueued for just those recipients who were deferred, and
3. The original message file is removed from the processing queue.

A message can be forcibly deferred, without regard to the dispositions of the recipients, by passing a value of true (1) for the **defer** argument. When a message is deferred, either because **defer** is true or all recipients have a deferred disposition, then the value supplied with the **reason** argument will be placed in the message's delivery failure log. If a zero length string is supplied for that argument, then the deferral reason, if any, for the last deferred recipient address will be used. Should the message be returned as an undeliverable message by PMDF's message return system, a copy of the log will be included with the returned message.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. Message not dequeued.

PMDFdisposeChannelCounters

PMDFdisposeChannelCounters

Dispose of a list of channel counters.

PASCAL *status* = **PMDF_dispose_channel_counters** (*counters*)

argument information

Argument	Data type	Access	Mechanism
counters	counter pointer	read/write	reference

C *status* = **PMDFdisposeChannelCounters** (*counters*)

argument information

```
int PMDFdisposeChannelCounters(PMDF_channel_counters
**counters)
```

ARGUMENTS *header*
Pointer to list of channel counters returned by a previous call to `PMDFgetChannelCounters`.

DESCRIPTION `PMDFdisposeChannelCounters` should be called to dispose of a previously allocated list of channel counters created by `PMDFgetChannelCounters`.

RETURN VALUES

`PMDF__OK` Normal, successful completion.

PMDFdisposeHeader

Dispose of a header data structure.

PASCAL *status* = **PMDF_dispose_header** (*header*)

argument information

Argument	Data type	Access	Mechanism
header	header pointer	read/write	reference

C *status* = **PMDFdisposeHeader** (*header*)

argument information

```
int PMDFdisposeHeader(PMDF_hdr **header)
```

ARGUMENTS

header

Address of a header structure returned by a previous call to `PMDFreadHeader` or `PMDFaddHeaderLine`.

DESCRIPTION

`PMDFdisposeHeader` should be called to dispose of a previously allocated header structure created by `PMDFreadHeader` or `PMDFaddHeaderLine`.

RETURN VALUES

`PMDF__OK` Normal, successful completion.

PMDFdone

PMDFdone

Deallocate PMDF data structures and resources.

PASCAL *status* = **PMDF_done**

C *status* = **PMDFdone** ()

argument information

int PMDFdone()

ARGUMENTS *None.*

DESCRIPTION After finishing all processing, PMDFdone should be called. Processes which run indefinitely should not repeatedly call PMDFinitialize and PMDFdone. PMDFinitialize and PMDFdone should, generally, be called only once per program run.

To shutdown any active message dequeue or enqueue contexts, call PMDFdequeueEnd, PMDFenqueueMessage, or PMDFabortMessage prior to calling PMDFdone. If PMDFdone is called while dequeue or enqueue contexts are still active, then any messages associated with active dequeue contexts will be deferred for later processing and any messages associated with active enqueue contexts will be deleted.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFenqueueInitialize

Initialize PMDF for message enqueueing operations.

PASCAL *status* = **PMDF_enqueue_initialize**

C *status* = **PMDFenqueueInitialize** ()

argument information

```
int PMDFenqueueInitialize()
```

ARGUMENTS *None.*

DESCRIPTION PMDFenqueueInitialize is called to initialize PMDF for message enqueue processing. PMDFenqueueInitialize should only be called once, after calling PMDFinitialize.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFenqueueMessage

PMDFenqueueMessage

Submit a message to PMDF's message queues.

PASCAL *status* = **PMDF_enqueue_message** (*nq_context*)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference

C *status* = **PMDFenqueueMessage** (*nq_context*)

argument information

```
int PMDFenqueueMessage(PMDF_nq **nq_context)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

DESCRIPTION The final step in enqueueing a message is to call `PMDFenqueueMessage`. This call submits the message which was being composed and sends it on its way. Should an error occur, `PMDFgetErrorText` can be called to obtain further details about the error. Note that only temporary processing errors are reported (e.g., write errors to the disk occurred when creating the message file in the PMDF channel queue directory). When a permanent processing error occurs (e.g., message size exceeds site-imposed limits), PMDF automatically generates a non-delivery notification and sends it to the envelope "From:" address specified with `PMDFstartMessageEnvelope`. The non-delivery notification will show the address of each recipient address which failed with a permanent error.

If the message is successfully enqueued as indicated by a return value of `PMDF__OK` or `PMDF__NOOP`, then `PMDFenqueueMessage` deletes the message context and nils (zeros) the context context pointer. If, however, an error occurs, the message context is not deleted. In that case `PMDFabortMessage` should be called to properly dispose of the message context. A new message enqueue context can be created with `PMDFstartMessageEnvelope`. That is, the process of submitting another message can be started with a call to `PMDFstartMessageEnvelope`.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. Message not enqueued.
PMDF__FCRT	File create error. The message could not be placed in the PMDF message queues. This is typically due to insufficient privileges although other possibilities exist such as insufficient disk space. Message not enqueued; message context not deleted. Delete with <code>PMDFabortMessage</code> .
PMDF__NO	Message could not be delivered owing to temporary processing problems of some sort (e.g., insufficient disk space to store the queued message).
PMDF__NOOP	Message had no envelope "To:" addresses; its delivery was effected by simply deleting it.

PMDFgetAddressProperty

PMDFgetAddressProperty

Parse an address and return the requested address property.

PASCAL *status* = **PMDF_get_address_property**
 (*address, property, result, result_len*)

argument information

Argument	Data type	Access	Mechanism
address	descriptor	read	reference
property	integer	read	value
result	descriptor	read/write	reference
result_len	unsigned word	write	reference

C *status* = **PMDFgetAddressProperty**
 (*address, address_len, property, result, result_len*)

argument information

```
int PMDFgetAddressProperty(char *address,  
                           int  address_len,  
                           int  property,  
                           char *result,  
                           int  *result_len)
```

ARGUMENTS

address

The address to parse. Length of this string can not exceed `BIGALFA_SIZE` bytes.

address_len

Length in bytes of the address to parse.

property

The address property to return.

result

String to receive the address property. Must be at least `ALFA_SIZE` bytes in length for `PMDF_get_address_property` or `ALFA_SIZE+1` bytes for `PMDFgetAddressProperty`.

result_len

Length in bytes of the returned property. Callers using `PMDFgetAddressProperty` must, on input, supply the maximum length in bytes of **result**.

DESCRIPTION `PMDFgetAddressProperty` can be used to parse an address and return the desired property. Moreover, `PMDFgetAddressProperty` can be used to see if an address is syntactically legal and to clean up addresses with minor syntax problems. The former is accomplished by seeing if `PMDF__PARSE` is returned and the latter by requesting the `PMDF_PROP_PROPER` property.

The accepted values for **property** are shown below and refer to an address of the form

phrase <@*otherhost*:*user*@*host*> (*comment*)

Symbolic name	Value	Description
<code>PMDF_PROP_ADDRESS</code>	1	Address part, @ <i>otherhost</i> : <i>user</i> @ <i>host</i> , of the address
<code>PMDF_PROP_DOMAIN</code>	2	Domain part, <i>host</i> , of the address
<code>PMDF_PROP_FRIENDLY</code>	3	See description below
<code>PMDF_PROP_LOCAL</code>	4	Local part, <i>user</i> , of the address
<code>PMDF_PROP_PHRASE</code>	5	Phrase part, <i>phrase</i> , of the address, if any
<code>PMDF_PROP_PROPER</code>	6	Full address including any phrases and comments
<code>PMDF_PROP_ROUTE</code>	7	Source route part, @ <i>otherhost</i> :, of the address, if any

The `PMDF_PROP_FRIENDLY` property can be used to attempt to extract a human name from the address. When this property is requested, the following steps are used to determine the value to return:

1. If a RFC 822 phrase *phrase* is present, then return it, else
2. If at least one RFC 822 comment *comment* is present, then return the first one, else
3. If the local part *user* is not a RFC 1327 AVPL, then return the local part, else
4. If a string of the form `/pn=value/` is present in the local part, then replace any dots in *value* with spaces and return that, else
5. If a string of the form `/s=svalue/` is not present in the local part, then return the local part, else
6. If a string of the form `/g=gvalue/` is present in the local part then return *gvalue svalue*, otherwise
7. Return *svalue*.

Note that `PMDF_get_address_property` can only handle a single address of length up to but not exceeding `BIGALFA_SIZE` bytes. If more than one address is present in the string, then `PMDF__NO` will be returned. So, if the address is longer than `BIGALFA_SIZE` bytes or more than one address can be present, `PMDFaddressParseList` and `PMDFaddressGetProperty` should instead be used.

PMDFgetAddressProperty

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	Bad parameter supplied: invalid value for property . No result returned.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. No result returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No result returned.
PMDF__INVSTRDES	Invalid string descriptor for result : descriptor has an invalid value in its DSC\$B_CLASS field. No result returned.
PMDF__NO	Invalid address. No result returned.
PMDF__STRTRU	Supplied string was too long; result truncated to fit.

PMDFgetBlockSize

Obtain the size in bytes of a PMDF block.

PASCAL *block_size* = **PMDF_get_block_size**

C *block_size* = **PMDFgetBlockSize** ()

argument information

```
int PMDFgetBlockSize()
```

ARGUMENTS *None.*

DESCRIPTION PMDF measures message sizes in units of “blocks”. Units of blocks are used when recording message sizes in log files and when determining if a message is large enough to warrant being fragmented into smaller messages. By default, a block is 1024 bytes; however, sites can change this size of a block with the `BLOCK_SIZE` option in the PMDF option file.

RETURN VALUES

`block_size` The size in bytes of a PMDF block.

PMDFgetChannelCounters

Obtain accumulated counters for one or more channels.

PASCAL *status = PMDF_get_channel_counters*
 (*channel, counters, count*)

argument information

Argument	Data type	Access	Mechanism
channel	descriptor	read	reference
timeout	signed longword	read	value
counters	counters pointer	write	reference
count	signed longword	write	reference

C *status = PMDFgetChannelCounters*
 (*channel, channel_len, timeout, counters, count*)

argument information

```
int PMDFgetChannelCounters(char *channel,
                             int channel_len,
                             int timeout,
                             PMDF_channel_counters **counters,
                             int *count)
```

ARGUMENTS

channel

String containing the name of the channel to retrieve counters for. The name can contain wild card characters. Length of the string, in bytes, can not exceed CHANLENGTH.

channel_len

Length in bytes of the channel name.

timeout

Maximum time, in seconds, to wait for counters to be synchronized.

counters

Pointer to list of channel counters. The format of each entry in the list is described in the Description section below.

count

Count of the number of channels for which counters have been returned.

DESCRIPTION PMDF accumulates in the form of counters message traffic statistics for each of its channels. These statistics, referred to as “channel counters”, correspond to those used by the Mail Monitoring MIB (RFC 1566) with a PMDF channel representing a “group” as defined by RFC 1566. The `PMDFgetChannelCounters` routine can be used to read these counters for one or more channels.

To obtain counters for more than one channel at a time, use wild cards in the channel name. For instance, to obtain counters for all TCP/IP channels use the name “*tcp*”. Similarly, to obtain counters for all channels, use the name “*”.

The counters are returned as a list pointed at by the **counters** argument. The list should be disposed of with the `PMDFdisposeChannelCounters` routine.

Each entry in the list has the structure

```
#define CHANLENGTH 32
typedef struct PMDF_channel_counters_s {
    char                channel_name[CHANLENGTH+1];
    int                 received_messages;
    int                 submitted_messages;
    int                 stored_messages;
    int                 delivered_messages;
    int                 received_volume;
    int                 submitted_volume;
    int                 stored_volume;
    int                 delivered_volume;
    int                 received_recipients;
    int                 submitted_recipients;
    int                 stored_recipients;
    int                 delivered_recipients;
    struct PMDF_channel_counters_s *next;
    int                 rejected_messages;
    int                 failed_messages;
    int                 attempted_messages;
    int                 rejected_volume;
    int                 failed_volume;
    int                 attempted_volume;
    int                 rejected_recipients;
    int                 failed_recipients;
    int                 attempted_recipients;
    int                 delivered_first_messages;
    int                 delivered_first_queue_count;
    int                 delivered_first_queue_time;
    int                 delivered_queue_count;
    int                 delivered_queue_time;
} PMDF_channel_counters;
```

This structure is predeclared as `PMDF_channel_stats` in the C `apidef.h` header and Pascal `apidef.pas` environment files. With the exception of the `channel_name` and `next` fields, each field is a long, signed integer value. The `channel_name` field is `CHANLENGTH+1` bytes long and gives the name of the channel corresponding to the counters in the entry. The `next` field is a pointer to

PMDFgetChannelCounters

another list entry. The end of the list is signified by a `next` field with a zero (nil) value.

The interpretation of each field is given in the Table 1–5.

Table 1–5 Channel Counters List Entry

Field name	Type	Description
<code>channel_name</code>	string	The name of the channel stored in a <code>CHANLENGTH+1</code> byte long string; <code>PMDFgetChannelCounters</code> will zero terminate the string.
<code>received_messages</code>	signed longword	The cumulative count of messages enqueued to the channel.
<code>submitted_messages</code>	signed longword	The cumulative count of messages enqueued by the channel.
<code>stored_messages</code>	signed longword	The current count of messages stored for the channel
<code>delivered_messages</code>	signed longword	The cumulative count of messages dequeued by the channel.
<code>received_volume</code>	signed longword	The cumulative volume of messages enqueued to the channel.
<code>submitted_volume</code>	signed longword	The cumulative volume of messages enqueued by the channel.
<code>stored_volume</code>	signed longword	The current volume of messages stored for the channel.
<code>delivered_volume</code>	signed longword	The cumulative volume of messages dequeued by the channel.
<code>received_recipients</code>	signed longword	The cumulative count of recipients specified in all messages enqueued to the channel.
<code>submitted_recipients</code>	signed longword	The cumulative count of recipients specified in all messages enqueued by the channel.
<code>stored_recipients</code>	signed longword	The current count of recipients specified in all messages currently stored for the channel.
<code>delivered_recipients</code>	signed longword	The cumulative count of recipients specified in all messages dequeued by the channel.
<code>next</code>	pointer	Pointer to the next list entry of channel counters.
<code>rejected_messages</code>	signed longword	The cumulative count of messages which, upon trying to be enqueued to the channel, were rejected.
<code>failed_messages</code>	signed longword	The cumulative count of messages enqueued to the channel which, when processed, failed to be delivered for one or more recipients owing to permanent errors of some sort (e.g., invalid recipient address).

Note: All volumes are measured in units of PMDF blocks. A PMDF block is, by default, 1024 bytes. However, this size can vary from system to system. The size of a PMDF block is controlled with the `BLOCK_SIZE` PMDF option. The `PMDFgetBlockSize` routine can be used to determine the current size of a PMDF block; *i.e.*, the setting of the `BLOCK_SIZE` option.

Table 1–5 (Cont.) Channel Counters List Entry

Field name	Type	Description
attempted_messages	signed longword	The cumulative count of messages enqueued to the channel whose delivery has been attempted.
rejected_volume	signed longword	The cumulative volume of messages which, upon trying to be enqueued to the channel, were rejected.
failed_volume	signed longword	The cumulative volume of messages enqueued to the channel which, when processed, failed to be delivered for one or more recipients owing to permanent errors of some sort (e.g., invalid recipient address).
attempted_volume	signed longword	The cumulative volume of messages enqueued to the channel whose delivery has been attempted.
rejected_recipients	signed longword	The cumulative count of recipient addresses which, upon trying to be enqueued to the channel, were rejected.
failed_recipients	signed longword	The cumulative count of recipients enqueued to the channel which, when processed, failed to be delivered owing to permanent errors of some sort (e.g., invalid recipient address).
attempted_recipients	signed longword	The cumulative count of recipients enqueued to the channel whose delivery has been attempted.
delivered_first_messages	signed longword	The cumulative count of messages enqueued to the channel which were successfully delivered (or returned as undeliverable) on their first processing attempt.
delivered_first_queue_count	signed longword	Cumulative count of first message delivery attempts made by the channel. When this value is less than <code>received_messages</code> , it means that delivery has not yet been attempted for all received messages. This is not unusual: this value is expected to lag behind <code>received_messages</code> .
delivered_first_queue_time	signed longword	Cumulative count of elapsed seconds between when a message is enqueued and when processing of its first delivery attempt completes. The result of dividing <code>delivered_first_queue_time</code> by <code>delivered_first_queue_count</code> gives the average amount of time in seconds spent by a message in the processing queues as it awaits its initial delivery attempt.
delivered_queue_count	signed longword	Cumulative count of message delivery attempts made by the channel.

Note: All volumes are measured in units of PMDF blocks. A PMDF block is, by default, 1024 bytes. However, this size can vary from system to system. The size of a PMDF block is controlled with the `BLOCK_SIZE` PMDF option. The `PMDFgetBlockSize` routine can be used to determine the current size of a PMDF block; *i.e.*, the setting of the `BLOCK_SIZE` option.

PMDFgetChannelCounters

Table 1–5 (Cont.) Channel Counters List Entry

Field name	Type	Description
<code>delivered_queue_time</code>	signed longword	Cumulative count of elapsed seconds between when a message is enqueued and when it is finally removed from the channel queue. The result of dividing <code>delivered_queue_time</code> by <code>delivered_queue_count</code> gives the average amount of time in seconds spent by a message in the processing queues.

Note: All volumes are measured in units of PMDF blocks. A PMDF block is, by default, 1024 bytes. However, this size can vary from system to system. The size of a PMDF block is controlled with the `BLOCK_SIZE` PMDF option. The `PMDFgetBlockSize` routine can be used to determine the current size of a PMDF block; *i.e.*, the setting of the `BLOCK_SIZE` option.

The **timeout** argument specifies the maximum time, in seconds, to wait for node-specific caches of counters to be synchronized with the cluster-wide database of counters. If the time limit is exceeded, then the cluster-wide counters will be returned as is, not necessarily up-to-date. Specify a value of zero to avoid waiting at all or a value of -1 to wait without a timeout.

The **timeout** argument has no effect on UNIX systems at present.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for channel : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. No counters returned.
<code>PMDF__NO</code>	Cannot access the counters; an interlock could not be obtained after ten attempts. No counters returned.
<code>PMDF__STRTRUERR</code>	Supplied channel name was too long; no counters returned.

PMDFgetChannelName

Obtain the name of the channel to be processed.

PASCAL *status* = **PMDF_get_channel_name**
 (*channel*, *channel_len*, *keywords1*, *reserved*)

argument information

Argument	Data type	Access	Mechanism
channel	descriptor	read/write	reference
channel_len	unsigned word	write	reference
keywords1	unsigned longword	write	reference
reserved	unsigned longword	write	reference

C *status* = **PMDFgetChannelName**
 (*channel*, *channel_len*, *keywords1*, *reserved*)

argument information

```
int PMDFgetChannelName(char      *channel,
                        int       *channel_len,
                        unsigned int *keywords1,
                        unsigned int *reserved)
```

ARGUMENTS

channel
 String to receive the channel name. Must be at least CHANLENGTH bytes in length for `PMDF_get_channel_name` or CHANLENGTH+1 bytes for `PMDFgetChannelName`.

channel_len
 Length in bytes of the returned channel name. Callers using `PMDFgetChannelName` must, on input, supply the maximum length in bytes of **channel**.

keywords1
 Unsigned longword of bit flags describing various channel options set with channel keywords. This argument can be omitted.

reserved
 Unsigned longword argument reserved for future use. Not used at present. This argument can be omitted.

PMDFgetChannelName

DESCRIPTION Channel programs typically service one or more instances of a channel, each instance having a distinct name which is specified in the PMDF configuration file. For example, a master (outbound) PhoneNet over DECnet channel program services all channels with names of the form `dn_x` where `x` distinguishes between each instance of a master PhoneNet over DECnet channel (*e.g.*, `dn_node1`, `dn_node2`, *etc.*). The routine `PMDFgetChannelName` can be used by channel programs to determine which instance of a channel they are servicing; *i.e.*, to determine the name of the particular channel they are processing.

Channel programs which enqueue mail or can return messages typically need to know the name of the particular channel they are processing. This channel name is then used when `PMDFstartMessageEnvelope` or `PMDFreturnMessage` is called.

In some cases, it can be necessary to “hard-code” a channel name into a program or otherwise obtain the channel name by a means other than `PMDFgetChannelName`. For instance, the channel name for TCP/IP slave channels is specified at compile time, and PhoneNet slave channels prompt for the name of the channel they are to process. In such cases, `PMDFgetChannelName` should not be used.

When specified, bits in the optional **keywords1** argument will be set as follows:

Bits	Usage
0, 1	These two bits specify whether or not the <code>headerbottom</code> , <code>headerinc</code> , or <code>headeromit</code> channel keywords have been specified: Bit 0 = 0, bit 1 = 0. <code>headeromit</code> : discard the message's header. Bit 0 = 1, bit 1 = 0. <code>headerinc</code> : preserve the message header and keep it at the top of the message. This is the default case when none of these three channel keywords have been applied to the channel. Bit 0 = 0, bit 1 = 1. <code>headerbottom</code> : preserve the message header but place it at the bottom of the message.
2	When set, indicates that the <code>master_debug</code> keyword was specified for this channel.
3	When set, indicates that the <code>slave_debug</code> keyword was specified for this channel.
4, 5	These two bits specify whether or not the <code>exquota</code> , <code>holdexquota</code> , or <code>noexquota</code> channel keywords have been specified: Bit 4 = 0, bit 5 = 0. <code>noexquota</code> : return the message as undeliverable if the recipient is over quota. Bit 4 = 1, bit 5 = 0. <code>holdexquota</code> : defer delivery of the message if the recipient is over quota. Bit 4 = 0, bit 5 = 1. <code>exquota</code> : deliver the message to the recipient even if they are over quota.

Bit 0 is the least significant bit.

Note that all other channel keywords which can be applied to the channel are automatically handled by PMDF.

On OpenVMS systems, the actual name of the particular channel being processed is specified by the logical `PMDF_CHANNEL`. The translation value of this logical gives the name of the channel being processed. On UNIX systems, the `PMDF_CHANNEL`

environment variable is instead used with the equivalence value of the variable being the name of the channel.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. Channel name not returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. Channel name not returned.
PMDF__INVSTRDES	Invalid string descriptor for channel : descriptor has an invalid value in its DSC\$B_CLASS field. Channel name not returned.
PMDF__NOCHANNEL	Either the channel name cannot be determined, or the channel cannot be located in the configuration file.
PMDF__STRTRU	Supplied string was too long; value truncated to fit.

PMDFgetDateTime

PMDFgetDateTime

Obtain the current date and time in an RFC 822/1123 compliant format.

PASCAL *status* = **PMDF_get_date_time**
 (*datetime*, *datetime_len*)

argument information

Argument	Data type	Access	Mechanism
<i>datetime</i>	descriptor	read/write	reference
<i>datetime_len</i>	unsigned word	write	reference

C *status* = **PMDFgetDateTime** (*datetime*, *datetime_len*)

argument information

```
int PMDFgetDateTime(char *datetime, int *datetime_len)
```

ARGUMENTS

datetime

String to receive the formatted date and time. Must be at least 27+N bytes long where N is the length of the local time zone string.

datetime_len

Length in bytes of the returned time string. Callers using `PMDFgetDateTime` must, on input, supply the maximum length in bytes of ***datetime***.

DESCRIPTION

The routine `PMDFgetDateTime` can be used to obtain the system's current date and time. The returned string will be in a format compatible with RFC 822 and RFC 1123; *e.g.*, "Sat, 04 May 2012 18:04:00 EDT". This string is then suitable for use in a "Date:" header line.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. Date and time not returned.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. Date and time not returned.

PMDF__INVSTRDES

Invalid string descriptor for **datetime**: descriptor has an invalid value in its DSC\$_CLASS field. Date and time not returned.

PMDF__STRTRU

Supplied string was too long; value truncated to fit.

PMDFgetEnvelopeId

Obtain the envelope id associated with this message.

PASCAL *status = PMDF_get_envelope_id*
 (*dq_context, envelope_id, envelope_id_len*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>envelope_id</i>	descriptor	read/write	reference
<i>envelope_id_len</i>	unsigned word	write	reference

C *status = PMDFgetEnvelopeId*
 (*dq_context, envelope_id, envelope_id_len*)

argument information

```
int PMDFgetEnvelopeId(PMDF_dq **dq_context,  
                      char *envelope,  
                      int *envelope_id_len)
```

ARGUMENTS *dq_context*
A message dequeue context created with `PMDFdequeueInitialize`.

envelope_id
String to receive the message's envelope id. Length must be at least `ALFA_SIZE+1` bytes.

envelope_id_len
Length in bytes of the envelope id. Callers using `PMDFgetEnvelopeId` must, on input, supply the maximum length in bytes of ***envelope_id***.

DESCRIPTION Messages queued to PMDF often carry with them two identification strings – “id’s” for short. The first is the “message id” as seen in the message’s RFC 822 “Message-id:” header line. This id is the same for all copies of a given message. The second id is the envelope id. Each copy of the message has a distinct envelope id, if it has any envelope id at all.

It is important to note that not all messages can have envelope id’s. Specifically, RFC 1891 forbids adding an envelope id to a message obtained via SMTP without an envelope id. As such, it is possible to find messages in PMDF’s queues

which have no envelope id's—these are messages which were received without an envelope id.

When a message dequeue is initiated, the message and envelope id's can be obtained by calling `PMDFgetEnvelopeId` and `PMDFgetMessageId`. It is particularly important to obtain the envelope id as it should be propagated forward by channels which re-enqueue the message for subsequent processing.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. No envelope id retrieved.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No envelope id retrieved.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. No envelope id retrieved.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for envelope_id : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. No envelope id retrieved.
<code>PMDF__STRTRU</code>	Supplied string was too long; value truncated to fit.

PMDFgetErrorText

Obtain any error message associated with a `PMDF__` error status code.

PASCAL *status* = **PMDF_get_error_text**
 (*nq_context*, *text*, *text_len*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>text</i>	descriptor	read/write	reference
<i>text_len</i>	unsigned word	write	reference

C *status* = **PMDFgetErrorText**
 (*nq_context*, *text*, *text_len*)

argument information

```
int PMDFgetErrorText(PMDF_nq **nq_context,  
                    char      *text,  
                    int       *text_len)
```

ARGUMENTS

text

String to receive a description associated with an error message. Must be at least `ALFA_SIZE+1` bytes in length.

text_len

Length in bytes of the returned description. Callers using `PMDFgetErrorText` must, on input, supply the maximum length in bytes of ***text***.

DESCRIPTION

In some cases, after a `PMDF__` error has been returned, additional information about the error can be obtained by calling `PMDFgetErrorText`. This additional information is returned as a text string and is suitable for writing to a log file. The applicable cases are

- after an error from `PMDFaddRecipient`,
- after an error from `PMDFenqueueMessage`, or
- after an error from `PMDFstartMessageEnvelope`.

The above cases do not include errors associated with bad call arguments; that is, do not apply when the error resulted from passing a bad parameter to the routine which returned the error.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. Error text not returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. Error text not returned.
PMDF__INVSTRDES	Invalid string descriptor for text : descriptor has an invalid value in its DSC\$_CLASS field. Error text not returned.
PMDF__STRTRU	Supplied string was too long; value truncated to fit.

PMDFgetHostName

PMDFgetHostName

Obtain the official local host name.

PASCAL *status* = **PMDF_get_host_name** (*host*, *host_len*)

argument information

Argument	Data type	Access	Mechanism
<i>host</i>	descriptor	read/write	reference
<i>host_len</i>	unsigned word	write	reference

C *status* = **PMDFgetHostName** (*host*, *host_len*)

argument information

```
int PMDFgetHostName(char *host, int *host_len)
```

ARGUMENTS

host

String to receive the official local host name. This string should be at least ALFA_SIZE+1 bytes long.

host_len

Length in bytes of the returned host string. Callers using `PMDFgetHostName` must, on input, supply the maximum length in bytes of ***host***.

DESCRIPTION

The official name of the local host (*i.e.*, the host name associated with the local, l, channel) can be obtained by calling `PMDFgetHostName`. This host name is typically used when constructing return addresses for local users. Such a return address is simply *user@host* where *user* is the name of the local user and *host* is the local host name.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. Host name not returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. Host name not returned.

PMDF__INVSTRDES

Invalid string descriptor for **host**: descriptor has an invalid value in its DSC\$B_CLASS field. Host name not returned.

PMDF__STRTRU

Supplied string was too long; value truncated to fit.

PMDFgetMessage

Access the next message in the message queue being processed.

PASCAL *status* = **PMDF_get_message**
 (*dq_context*, *filename*, *filename_len*, *from*, *from_len*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>filename</i>	descriptor	read/write	reference
<i>filename_len</i>	unsigned word	write	reference
<i>from</i>	descriptor	read/write	reference
<i>from_len</i>	unsigned word	write	reference

C *status* = **PMDFgetMessage**
 (*dq_context*, *filename*, *filename_len*, *from*, *from_len*)

argument information

```
int PMDFgetMessage(PMDF_dq **dq_context,  
                  char      *filename,  
                  int       *filename_len,  
                  char      *from,  
                  int       *from_len)
```

ARGUMENTS

dq_context

A message dequeue context created with `PMDFdequeueInitialize`.

filename

String to receive the name of the file containing the accessed message. Length must be at least `ALFA_SIZE+1` bytes.

filename_len

Length in bytes of the returned file name. Callers using `PMDFgetMessage` must, on input, supply the maximum length in bytes of ***filename***.

from

String to receive the envelope "From:" address of the accessed message. Length must be at least `ALFA_SIZE+1` bytes.

from_len

Length in bytes of the envelope "From:" address. Callers using `PMDFgetMessage` must, on input, supply the maximum length in bytes of ***from***.

DESCRIPTION PMDFgetMessage should be called repeatedly to access, one at a time, each message requiring processing. Each message to be processed will only be presented once; *i.e.*, a job will not repeatedly see a message which it has deferred. When PMDFgetMessage returns the status code PMDF__EOF, no more messages remain to be processed.

The returned envelope "From:" address should be saved as it can be needed if the program either enqueues a new message or returns the accessed message. The returned file name can usually be ignored as the API routines manage all access to the message file including opening the file, reading it, closing it, and deleting it when it is dequeued.

A message accessed with PMDFgetMessage can be processed using any of the routines accepting a **dq_context** argument.

After processing an accessed message, the message should de-accessed with PMDFdequeueMessageEnd.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__EOF	No more messages to be processed.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. Message not accessed.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. Message not accessed.
PMDF__INVSTRDES	Invalid string descriptor for filename or from : one or both of the descriptors has an invalid value in its DSC\$B_CLASS field.
PMDF__STRTRU	Supplied filename or from string was too long; value truncated to fit.

PMDFgetMessageId

PMDFgetMessageId

Obtain the message id associated with this message.

PASCAL *status* = **PMDF_get_message_id**
 (*dq_context*, *message_id*, *message_id_len*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>message_id</i>	descriptor	read/write	reference
<i>message_id_len</i>	unsigned word	write	reference

C *status* = **PMDFgetMessageId**
 (*dq_context*, *message_id*, *message_id_len*)

argument information

```
int PMDFgetMessageId(PMDF_dq **dq_context,  
                    char      *message,  
                    int       *message_id_len)
```

ARGUMENTS *dq_context*
A message dequeue context created with `PMDFdequeueInitialize`.

message_id
String to receive the message's message id. Length must be at least `ALFA_SIZE+1` bytes.

message_id_len
Length in bytes of the message id. Callers using `PMDFgetMessageId` must, on input, supply the maximum length in bytes of ***message_id***.

DESCRIPTION The `PMDFgetMessageId` routine provides ready access to a message's message id. Note that the message id can also be obtained by reading and parsing the message's header. However, for efficiency purposes, PMDF stores a copy of the message id in the message envelope. This routine provides access to that envelope copy.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. No message id retrieved.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. No message id retrieved.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No message id retrieved.
PMDF__INVSTRDES	Invalid string descriptor for message_id : descriptor has an invalid value in its DSC\$B_CLASS field. No message id retrieved.
PMDF__STRTRU	Supplied string was too long; value truncated to fit.

PMDFgetPostmasterAddress

PMDFgetPostmasterAddress

Obtain the local postmaster's address.

PASCAL *status* = **PMDF_get_postmaster_address**
 (*address*, *address_len*)

argument information

Argument	Data type	Access	Mechanism
address	descriptor	read/write	reference
address_len	unsigned word	write	reference

C *status* = **PMDFgetPostmasterAddress**
 (*address*, *address_len*)

argument information

```
int PMDFgetPostmasterAddress(char *address, int *address_len)
```

ARGUMENTS **address**
String to receive the local postmaster's address. Length must be at least ALFA_SIZE+1 bytes.

address_len
Length in bytes of the postmaster's address. Callers using `PMDFgetPostmaster-address` must, on input, supply the maximum length in bytes of **address**.

DESCRIPTION `PMDFgetPostmasterAddress` can be used to obtain the mail address for the local postmaster. Note, however, that it usually is not a good idea for programs to send mail to the postmaster. In many situations, sending mail to the postmaster when failures occur can lead to mail loops; *e.g.*, the mail sent to the postmaster itself fails and generates a message to the postmaster which then fails and generates yet another message to the postmaster, *ad infinitum*.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No address returned.

PMDFgetPostmasterAddress

PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No address returned.
PMDF__INVSTRDES	Invalid string descriptor for address : descriptor has an invalid value in its DSC\$B_CLASS field. No address returned.
PMDF__STRTRU	Supplied string was too long; value truncated to fit.

PMDFgetRecipient

PMDFgetRecipient

Obtain the next envelope "To:" address from a message.

PASCAL *status* = **PMDF_get_recipient**
 (*dq_context*, *address*, *address_len*, *orig_address*,
 orig_address_len)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>address</i>	descriptor	read/write	reference
<i>address_len</i>	unsigned word	write	reference
<i>orig_address</i>	descriptor	read/write	reference
<i>orig_address_len</i>	unsigned word	write	reference

C *status* = **PMDFgetRecipient**
 (*dq_context*, *address*, *address_len*, *orig_address*,
 orig_address_len)

argument information

```
int PMDFgetRecipient(PMDF_dq **dq_context,  
                    char *address,  
                    int *address_len,  
                    char *orig_address,  
                    int *orig_address_len)
```

ARGUMENTS

dq_context

A message dequeue context created with `PMDFdequeueInitialize`.

address

String to receive an envelope "To:" address from the message's envelope. Length must be at least `ALFA_SIZE+1` bytes.

address_len

Length in bytes of the envelope "To:" address. Callers using `PMDFgetRecipient` must, on input, supply the maximum length in bytes of ***address***.

orig_address

String to receive the original form of the envelope "To:" address, if known. Length must be at least `ALFA_SIZE+1` bytes.

orig_address_len

Length in bytes of the original envelope "To:" address, if known. Callers using PMDFgetRecipient must, on input, supply the maximum length in bytes of **orig_address**.

DESCRIPTION

PMDFgetRecipient should be called repeatedly to obtain each envelope "To:" address from a message. In each call to PMDFgetRecipient a single envelope "To:" address will be returned in **address**. After all addresses have been output, PMDFgetRecipient will return the status code PMDF__EOF. For example, if the message envelope has two "To:" addresses, then three calls to PMDFgetRecipient should be made. In the first two calls, the two addresses will be output along with the return status code PMDF__OK. In the third call no address will be output and the status code PMDF__EOF will be returned.

After each call in which PMDFgetRecipient returns PMDF__OK, a call should be made to PMDFgetRecipientFlags to obtain the NOTARY processing flags associated with the envelope "To:" address.

After all of the envelope "To:" addresses have been read, the message header and body can be read with PMDFreadHeader, PMDFreadLine, and PMDFreadText.

The **orig_address** gives, if known, the original form of the envelope "To:" address. This is original address is carried with the message and used when generating notifications concerning the message. When calling PMDFrecipientDisposition or re-enqueuing a message to PMDF, this original address should be supplied.

After the channel processes an envelope "To:" address and determines its disposition, PMDFrecipientDisposition must be called. The NOTARY flag obtained with PMDFgetRecipientFlags for that address as well the original form of the address must be supplied to PMDFrecipientDisposition. By supplying this disposition information, PMDF can automatically generate determine whether or not the message needs to be deferred for later processing and generate any required notifications when the message being dequeued is de-accessed.

If the status code PMDF__NO is returned, then the message file was found to be missing both a message header and message body and has been deleted. The calling program should abort processing of the current message and call either PMDFdequeueMessageEnd with the **defer** argument set to true.

If the status code PMDF__STRTRU is returned, then it is probably not safe to proceed: the envelope "To:" address had to be truncated to fit into the supplied address string and a truncated address is generally worthless.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. No address retrieved.
PMDF__EOF	No more envelope To: addresses.

PMDFgetRecipient

PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. No address retrieved.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No address retrieved.
PMDF__INVSTRDES	Invalid string descriptor for address : descriptor has an invalid value in its DSC\$B_CLASS field. No address retrieved.
PMDF__NO	Accessed message file was corrupt. It has been deleted. Abort current dequeue processing by calling PMDFdequeueMessageEnd. No address retrieved.
PMDF__STRTRU	Supplied string was too long; value truncated to fit.

PMDFgetRecipientFlags

Obtain the NOTARY flags for the previously obtained envelope recipient address.

PASCAL *status* = **PMDF_get_recipient_flags**
 (*dq_context*, *flags*)

argument information

Argument	Data type	Access	Mechanism
dq_context	context pointer	read/write	reference
flags	integer	write	reference

C *status* = **PMDFgetRecipientFlags**
 (*dq_context*, *notary_flags*)

argument information

```
int PMDFgetRecipientFlags(PMDF_dq **dq_context,
int *notary_flags)
```

ARGUMENTS *dq_context*
 A message dequeue context created with `PMDFdequeueInitialize`.

notary_flags
 Longword integer to receive the NOTARY flag bits.

DESCRIPTION PMDF mail messages carry per recipient NOTARY information in their envelope. This information is aligned with the NOTARY SMTP extension as described in RFC 1891 and describes failure and success handling requested by the sender (e.g., send a delivery receipt, send failure notifications but do not include return of content, never send any form of notifications, etc.).

When dequeuing a message, every time `PMDFgetRecipient` is called and returns `PMDF__OK`, `PMDFgetRecipientFlags` should immediately be called afterwards. The **notary_flags** value returned should then be saved and, once the disposition of the associated envelope recipient address is known, `PMDFrecipientDisposition` called with the recipient address, the value of **notary_flags**, and the disposition of the address.

notary_flags is a bit-encoded value. The interpretation of the individual bits are given in Table 1–6. These flags are based upon RFC 1891; refer to that document for details on their usage.

PMDFgetRecipientFlags

Table 1–6 Envelope To: Address NOTARY Flags

Symbolic name	Bit	Mask value	Description
PMDF_RECEIPT_HEADER	0	1	Include the message's header in notification messages concerning this envelope "To:" address. RFC 1891 equivalent: RET=HDRS.
PMDF_RECEIPT_NOHEADER	1	2	Do not include the message's header in notification messages concerning this envelope "To:" address. No RFC 1891 equivalent.
PMDF_RECEIPT_FAILURES	2	4	Send a non-delivery notification (NDN) to the envelope "From:" address if the message cannot be delivered to this envelope "To:" address. RFC 1891 equivalent: NOTIFY=FAILURE.
PMDF_RECEIPT_SUCCESSES	3	8	When the message is successfully delivered to this envelope "To:" address, send a delivery status notification (DSN) to the envelope "From:" address indicating successful delivery. RFC 1891 equivalent: NOTIFY=SUCCESS.
PMDF_RECEIPT_DELAYS	4	16	When delivery of the message to this envelope "To:" address is delayed for some period of time, send a delivery status notification (DSN) to the envelope "From:" address reporting the delay. RFC 1891 equivalent: NOTIFY=DELAY.
PMDF_RECEIPT_NEVER	6	64	Do not send back notification messages of any sort concerning this envelope "To:" address. RFC 1891 equivalent: NOTIFY=NEVER.

When gatewaying mail to another mail system, the NOTARY information should be converted to equivalent requests in the other mail system. If they cannot be, then a disposition of PMDF_DISP_RELAYED_FOREIGN should be set for the gatewayed envelope "To:" address.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. No flags retrieved.
PMDF__NO	Information not available.

PMDFgetUniqueString

Generate a unique, eighteen character string.

PASCAL *status* = **PMDF_get_unique_string** (*string*, *string_len*)

argument information

Argument	Data type	Access	Mechanism
<i>string</i>	descriptor	read/write	reference
<i>string_len</i>	unsigned word	write	reference

C *status* = **PMDFgetUniqueString** (*string*, *string_len*)

argument information

```
int PMDFgetUniqueString(char *string, int *string_len)
```

ARGUMENTS

string

String to receive the psuedo-random unique character string. Length must be at least 19 bytes.

string_len

Length in bytes of the unique string. Callers using `PMDFgetUniqueString` must, on input, supply the maximum length in bytes of the string buffer.

DESCRIPTION

`PMDFgetUniqueString` will return a psuedo-random character string composed of a fixed number of characters chosen from the thirty-six character alphabet 0, 1, 2, ..., 9, A, B, C, ..., Z. On OpenVMS systems, this string will be 18 characters long and unique cluster-wide (*i.e.*, no two calls made in the same cluster will generate the same string). On UNIX systems, the string is 14 characters long.

`PMDFgetUniqueString` is a useful utility for programs which need to generate, for instance, unique file names. Note that the generated string can begin with a numeral. Thus, on file systems which require that file names begin with a non-numeric character, a character such as a "A" should be prepended to the string to produce a valid file name. Truncating the string will compromise its uniqueness.

PMDFgetUniqueString

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. No string returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No string returned.
PMDF__INVSTRDES	Invalid string descriptor for string : descriptor has an invalid value in its DSC\$_CLASS field. No string returned.
PMDF__STRTRU	Supplied string was too short; value truncated to fit.

PMDFgetUserName

Determine the user name associated with the currently running process.

PASCAL *status* = **PMDF_get_user_name**
 (*user_name*, *user_name_len*)

argument information

Argument	Data type	Access	Mechanism
<i>user_name</i>	descriptor	read/write	reference
<i>user_name_len</i>	unsigned word	write	reference

C *status* = **PMDFgetUserName**
 (*user_name*, *user_name_len*)

argument information

```
int PMDFgetUserName(char *user_name, int *user_name_len)
```

ARGUMENTS *user_name*
 String to receive the current process's user name. Length must be sufficient to receive any user name supported by the operating system in use. Callers of **PMDFgetUserName** must include an extra byte for zero termination of the returned string.

user_name_len
 Length in bytes of the returned user name. Callers using **PMDFgetUserName** must, on input, supply the maximum length in bytes of **user_name**.

DESCRIPTION **PMDFgetUserName** can be called to determine the user name associated with the currently running process.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. No user name returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No user name returned.

PMDfgetUserName

PMDf__INVSTRDES

Invalid string descriptor for **user_name**: descriptor has an invalid value in its DSC\$B_CLASS field. No user name returned.

PMDf__STRTRU

Supplied string was too long; value truncated to fit.

PMDFinitialize

Initialize PMDF data structures and resources.

PASCAL *status* = **PMDF_initialize** (*ischannel*)

argument information

Argument	Data type	Access	Mechanism
ischannel	boolean	read	reference

C *status* = **PMDFinitialize** (*ischannel*)

argument information

```
int PMDFinitialize(int ischannel)
```

ARGUMENTS

ischannel

If true, then user-to-channel access checks will be disabled. If false, then user-to-channel access checks will be enabled.

DESCRIPTION

With the exception of `PMDFsetMutex`, `PMDFinitialize` must be called prior to calling any other API routines. This allocates and initializes internal data structures used by the API and PMDF. `PMDFinitialize` should only be called once. After all processing is completed, `PMDFdone` should be called to release any allocated memory, and ensure that any open files are properly closed.

The **ischannel** flag is used to enable or disable rightslist based user-to-channel access checks. Programs which enqueue messages in behalf of users (e.g., user agents), should invoke `PMDFinitialize` with **ischannel** false; channel programs which enqueue mail should invoke `PMDFinitialize` with **ischannel** true. When **ischannel** is false, `PMDFenqueueMessage` will also close the queue cache database after enqueueing a message. On OpenVMS systems, channel programs which run indefinitely (e.g., detached processes) should supply a call back procedure to `PMDFsetCallBack` so that when a PMDF CACHE/CLOSE command is issued the program can call `PMDFcloseQueueCache` when convenient. See Section 1.7 for a further discussion of this issue.

Multithreaded routines must call `PMDFsetMutex` prior to calling `PMDFinitialize`.

PMDFinitialize

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FOPN	Initialization failed. One or more PMDF configuration files could not be accessed. PMDF configuration files are incorrectly protected. Note that the use of <code>PMDFinitialize</code> does not require any privileges; unprivileged users should be able to invoke this particular routine.
PMDF__NO	Initialization failed owing to a version mismatch between the current version of PMDF and the sites compiled configuration. Either the PMDF configuration needs to be recompiled or the character set tables need to be recompiled.

PMDFlog

Write a line of text to the channel log file.

PASCAL *status* = **PMDF_log** (*text*, *time_stamp*)

argument information

Argument	Data type	Access	Mechanism
<i>text</i>	descriptor	read	reference
<i>time_stamp</i>	boolean	read	value

C *status* = **PMDFlog** (*text*, *text_len*, *time_stamp*)

argument information

```
int PMDFlog(char    *text,
               int    text_len,
               int    time_stamp)
```

ARGUMENTS

text

String of text to write to the log file. Cannot exceed a length of 65,535 bytes.

text_len

Length in bytes of ***text***.

time_stamp

When true, output a time stamp to the log file prior to writing out the text string.

DESCRIPTION

Channels written using the PMDF API should write output using `PMDFlog`. They should not, for instance, attempt to write to `stdout` or `stderr`. So doing will lead to the output going to unexpected places such as the job controller's log file or down a network connection.

The `PMDF_log` routine writes *text* to the correct output destination; *e.g.*, the channel's log file or the terminal if the channel is running interactively. If debugging has been enabled with `PMDFdebug`, then the output will go to the same destination as the PMDF debugging output.

When ***time_stamp*** is true, a time stamp will first be output. For example, the call

PMDFlog

```
PMDF_log("Resuming message processing", true);
```

would result in output similar to

```
04-MAY-2012 18:04:00: Resuming message processing
```

Note that the channel log file is distinct from the PMDF log file. The `PMDFcloseLogFile` routine closes the PMDF log file and not the channel log file.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for text : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. Text not written.
<code>PMDF__STRTRU</code>	Input string's length exceeded 65,535 bytes; only the first 65,535 bytes were output.

PMDFmappingApply

Pass an input string through a mapping table.

PASCAL *status* = **PMDF_mapping_apply**
 (*mapping, instr, outstr, outstr_len, flags, match*)

argument information

Argument	Data type	Access	Mechanism
mapping	signed longword	read	value
instr	descriptor	read	reference
outstr	descriptor	read/write	reference
outstr_len	unsigned word	write	reference
flags	descriptor	read/write	reference
match	boolean	write	reference

C *status* = **PMDFmappingApply**
 (*mapping, instr, instr_len, outstr, outstr_len, flags, match*)

argument information

```
int PMDFmappingApply(int mapping,
                    char *instr,
                    int instr_len,
                    char *outstr,
                    int *outstr_len,
                    unsigned char *flags,
                    int *match)
```

ARGUMENTS

- mapping***
Reference to a mapping table returned by PMDFmappingLoad.
- instr***
Input string to process with the specified mapping table. The length of the string can not exceed ALFA_SIZE bytes.
- instr_len***
Length in bytes of the input string, **instr**.
- outstr***
String to receive the output, if any, of the mapping. Must be at least ALFA_SIZE+1 bytes in length.

PMDFmappingApply

outstr_len

Length in bytes of the output of the mapping. Will be set to 0 if no output is produced. Callers using `PMDFmappingApply` must, on input, supply the maximum length in bytes of **outstr**.

flags

Bit array of length at least 32 bytes (256 bits) which, on output, will contain bit encoded information about the mapping process.

match

For `PMDFmappingApply`, a boolean indicating whether or not a match was found. For `PMDFmappingApply` an integer indicating whether or not a match was found. If true (1) a match was found; if false (0) no match was found.

DESCRIPTION

`PMDFmappingApply` is used to apply a previously loaded mapping table to an input string. Consult the *PMDF System Manager's Guide* for details on the use of mapping tables and the mapping file in which mapping tables reside.

If the input string matches an entry in the table, then the result of the mapping is returned in **outstr** and **match** set true. Otherwise, **match** will be false and **outstr_len** set to zero.

Applications can require that special sequences such as `$Y` or `$N` be used in mapping table templates. The presence of such sequences are indicated in the **flags** bit array. These sequences, called metacharacters, will not appear in the output string itself. The output string produced by a template with a `$Y` in it will not contain `$Y`. However, bit 89, the ordinal value of the ASCII character `Y`, will be set in **flags**.

The interpretation of the first 256 bits in **flags** are given in the table below. Bit 0 is the low-order bit of the first byte in **flags**, bit 7 is the high-order bit of that same byte, bit 8 is the low-order bit of the next byte, and so forth.

Bit	Description
0—31	For $0 \leq n \leq 31$, bit n set indicates that $n + 1$ matches occurred. When bit n is set, bits $n - 1$, $n - 2$, ..., 0 will also be set.
32	When bit 32 is set, 32 or more matches occurred.
33—255	When bit n , $33 \leq n \leq 255$, is set, then the two character sequence <code>\$x</code> appeared in the output string, where x is the ASCII character with ordinal index n . This sequence will not actually appear in the output string itself. Bits 36, 67, 99, 69, 101, 76, 108, 82, and 114 are never set; they correspond to the sequences <code>\$\$</code> , <code>\$C</code> , <code>\$c</code> , <code>\$E</code> , <code>\$e</code> , <code>\$L</code> , <code>\$l</code> , <code>\$R</code> , and <code>\$r</code> used by the mapping facilities.

To illustrate the usage of **flags**, consider the mapping table

SAMPLE-TABLE

1	2\$A\$R
2	3\$B

The input string 1 will match the first entry of the table, and produce the output string 2. Because of the `$R` metacharacter, the mapping will be reapplied using 2

as the new input string. When 2 is mapped, it will match the second entry and produce the output string 3. Now, when 1 is mapped with `PMDFmappingApply`, the final output string will be 3, and bits 0, 1, 65, and 66 of **flags** will be set. The first two bits indicate that two matches in the mapping table were made. Bits 65 and 66 indicate that the metacharacters \$A and \$B were encountered in the templates of those matching entries. If 2 is mapped with `PMDFmappingApply`, then the output string will again be 3, but **flags** will have only bits 0 and 66 set. If any other string is mapped, then no output string will be returned and no bits in **flags** will be set.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. Mapping result not returned.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. Channel name not returned.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for instr , outstr , or flags : one or more of the descriptors has an invalid value in its <code>DSC\$B_CLASS</code> field. Mapping not performed or results not returned.
<code>PMDF__STRTRU</code>	Mapping result was too long to fit into the supplied output string; result truncated to fit.
<code>PMDF__STRTRUERR</code>	Input string is too long or flags string too short. Mapping not performed.

PMDFmappingLoad

PMDFmappingLoad

Access a mapping table.

PASCAL `status = PMDF_mapping_load (table, mapping)`

argument information

Argument	Data type	Access	Mechanism
table	descriptor	read	reference
mapping	signed longword	write	reference

C `status = PMDFmappingLoad (table, table_len, mapping)`

argument information

```
int PMDFmappingLoad(char *table,
                    int table_len,
                    int *mapping)
```

ARGUMENTS

table

Name of the table to load. The length of the string can not exceed ALFA_SIZE bytes.

table_len

Length in bytes of the table name.

mapping

Reference to the loaded mapping table for use with PMDFmappingApply.

DESCRIPTION

Before a mapping table can be used to map input strings, it must first be loaded. This is accomplished with PMDFmappingLoad. Any number of tables can be loaded, one table per PMDFmappingLoad call. Once a mapping table is loaded, it can be used with PMDFmappingApply. There is no call to make to unload a table.

PMDFinitialize must be called prior to the first call to PMDFmappingLoad. Failure to initialize PMDF first will result in a PMDF__NO error.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__INVSTRDES	Invalid string descriptor for table : descriptor has an invalid value in its DSC\$B_CLASS field. No table loaded.
PMDF__NO	PMDFinitialize has not yet been called. No table loaded.
PMDF__NOMAPPING	Cannot load specified mapping table; check to see if the mapping table exists. No table loaded.
PMDF__STRTRUERR	Supplied table name is too long. No table loaded.

PMDFOptionDispose

PMDFOptionDispose

Dispose of an option file context.

PASCAL *status* = **PMDF_option_dispose** (*opt_context*)

argument information

Argument	Data type	Access	Mechanism
<i>opt_context</i>	context pointer	read	value

C *status* = **PMDFOptionDispose** (*opt_context*)

argument information

```
int PMDFOptionDispose(PMDF_opt *opt_context)
```

ARGUMENTS *opt_context*
Pointer to context information generated by a previous call to `PMDFOptionRead`.

DESCRIPTION `PMDFOptionDispose` should be called to dispose of a previously allocated option context created by `PMDFOptionread`. It is okay to pass in a zero (nil) value for ***opt_context***.

RETURN VALUES

`PMDF__OK` Normal, successful completion.

PMDFOptionGetInteger

Get the value of an integer-valued option from an option file.

PASCAL *status* = **PMDF_option_get_integer**
 (*opt_context*, *name*, *value*)

argument information

Argument	Data type	Access	Mechanism
opt_context	context pointer	read	value
name	descriptor	read	reference
value	signed longword	write	reference

C *status* = **PMDFOptionGetInteger**
 (*opt_context*, *name*, *name_len*, *value*)

argument information

```
int PMDFOptionGetInteger(PMDF_opt *opt_context,
                          char      *name,
                          int       name_len,
                          int       *value)
```

ARGUMENTS ***opt_context***
 Pointer to context information generated by a previous call to `PMDFOptionRead`.

name
 Name of the option to obtain the value of. Name can not exceed a length in bytes of `SHORT_ALFA`. Option names are treated as case insensitive strings.

name_len
 Length in bytes of the option name.

value
 Value of the specified option.

DESCRIPTION `PMDFOptionGetInteger` returns in **value** the value of the specified option. If the option was not specified in the option file or if **opt_context** is zero (nil), then the content of **value** is left unchanged.

PMDFOptionGetInteger

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__INVSTRDES	Invalid string descriptor for name : descriptor has an invalid value in its DSC\$B_CLASS field. No option value returned.
PMDF__STRTRUERR	Supplied name string exceeds the maximum permitted length. No option value returned.

PMDFOptionGetReal

Get the value of an single precision, floating point-valued option from an option file.

PASCAL *status* = **PMDF_option_get_real**
 (*opt_context*, *name*, *value*)

argument information

Argument	Data type	Access	Mechanism
opt_context	context pointer	read	value
name	descriptor	read	reference
value	single precision real	write	reference

C *status* = **PMDFOptionGetReal**
 (*opt_context*, *name*, *name_len*, *value*)

argument information

```
int PMDFOptionGetReal(PMDF_opt *opt_context,
                      char *name,
                      int name_len,
                      float *value)
```

ARGUMENTS ***opt_context***
 Pointer to context information generated by a previous call to `PMDFOptionRead`.

name
 Name of the option to obtain the value of. Name can not exceed a length in bytes of `SHORT_ALFA`. Option names are treated as case insensitive strings.

name_len
 Length in bytes of the option name.

value
 Value of the specified option.

DESCRIPTION `PMDFOptionGetReal` returns in **value** the value of the specified option. If the option was not specified in the option file or if **opt_context** is zero (nil), then the content of **value** is left unchanged.

PMDFOptionGetReal

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__INVSTRDES	Invalid string descriptor for name : descriptor has an invalid value in its DSC\$B_CLASS field. No option value returned.
PMDF__STRTRUERR	Supplied name string exceeds the maximum permitted length. No option value returned.

PMDFOptionGetString

Get the value of a string-valued option from an option file.

PASCAL *status* = **PMDF_option_get_string**
 (*opt_context*, *name*, *value*, *value_len*)

argument information

Argument	Data type	Access	Mechanism
<i>opt_context</i>	context pointer	read	value
<i>name</i>	descriptor	read	reference
<i>value</i>	descriptor	read/write	reference
<i>value_len</i>	unsigned word	write	reference

C *status* = **PMDFOptionGetString**
 (*opt_context*, *name*, *name_len*, *value*, *value_len*, *max_len*)

argument information

```
int PMDFoptionGetString(PMDF_opt *opt_context,
                        char *name,
                        int name_len,
                        char *value,
                        int *value_len,
                        int max_len)
```

ARGUMENTS

- opt_context***
 Pointer to context information generated by a previous call to `PMDFOptionRead`.
- name***
 Name of the option to obtain the value of. Name can not exceed a length in bytes of `SHORT_ALFA`. Option names are treated as case insensitive strings.
- name_len***
 Length in bytes of the option name.
- value***
 Value of the specified option. String must be large enough to receive at least `BIGALFA_SIZE+1` bytes.
- value_len***
 Length in bytes of the returned value.

PMDFoptionGetString

max_len

The maximum length in bytes of **value**, not including any NULL terminator.

DESCRIPTION PMDFoptionGetString returns in **value** the value of the specified option. If the option was not specified in the option file or if **opt_context** is zero (nil), then the content of **value** and **value_len** is left unchanged.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__FATERRLIB	Call to LIB\$SCOPY_R_DX failed owing to a fatal internal error in the OpenVMS Run Time Library. No value returned.
PMDF__INSVIRMEM	Insufficient virtual memory: call to LIB\$GET_VM made by LIB\$SCOPY_R_DX has failed. No value returned.
PMDF__INVSTRDES	Invalid string descriptor for either name or value or both: descriptor for one or both has an invalid value in its DSC\$B_CLASS field. No option value returned.
PMDF__STRTRU	Supplied value string was too short. Option value truncated to fit in value .
PMDF__STRTRUERR	Supplied name string exceeds the maximum permitted length. No option value returned.

PMDFOptionRead

Read an option file.

PASCAL `status = PMDF_option_read (opt_context, filename)`

argument information

Argument	Data type	Access	Mechanism
opt_context	context pointer	write	reference
filename	descriptor	read	reference

C `status = PMDFOptionRead (opt_context, filename, filename_len)`

argument information

```
int PMDFOptionRead(PMDF_opt **opt_context,
                  char *filename,
                  int filename_len)
```

ARGUMENTS

opt_context

Pointer to context information generated by PMDFOptionRead.

filename

Full file specification specifying the option file to read. Length can not exceed ALFA_SIZE bytes.

filename_len

Length in bytes of the filename.

DESCRIPTION

PMDFOptionRead is used to read PMDF-style option files. The values for options can then be obtained using the PMDFOptionGetInteger, PMDFOptionGetReal, and PMDFOptionGetString routines. When finished obtaining option values, dispose of the **opt_context** with a call to PMDFOptionDispose.

Note that when no option file exists or the file contains no entries, the returned value for **opt_context** will be zero (nil). It is okay to pass a zero value for **opt_context** to the other routines which accept **opt_context**. This allows a program to blindly call the various option routines without regard to whether or not an option file exists.

PMDFoptionRead

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__DONE	Normal, successful completion. No option file existed.
PMDF__INVSTRDES	Invalid string descriptor for filename : descriptor has an invalid value in its DSC\$B_CLASS field. Option file not processed.
PMDF__NO	Error reading option file; most likely means that there is a syntax error in the option file.
PMDF__STRTRUERR	Supplied filename string exceeds the maximum permitted length. Option file not processed.

PMDFqueueCacheEnd

Dispose of a queue cache database context.

PASCAL *status* = **PMDF_queue_cache_end** (*cache_context*)

argument information

Argument	Data type	Access	Mechanism
cache_context	context pointer	read/write	reference

C *status* = **PMDFqueueCacheEnd** (*cache_context*)

argument information

```
void PMDFqueueCacheEnd(PMDF_qc **cache_context)
```

ARGUMENTS *cache_context*
Queue cache read context created with PMDFqueueCacheGetEntry.

DESCRIPTION Normally, queue cache contexts generated with PMDFqueueCacheGetEntry are automatically disposed of by PMDFqueueCacheGetEntry when it returns PMDF__EOM. To prematurely dispose of a context, use PMDFqueueCacheEnd.

RETURN VALUES

PMDF__OK Normal, successful completion.

PMDFQueueCacheGetEntry

Retrieve an entry from the queue cache database.

PASCAL *status = PMDF_queue_cache_get_entry*
 (*cache_context, item_list, reserved1, reserved2*)

argument information

Argument	Data type	Access	Mechanism
cache_context	context pointer	read/write	reference
item_list	item list	read/write	reference
reserved1	descriptor	read	reference
reserved2	descriptor	read	reference

C *status = PMDFQueueCacheGetEntry*
 (*cache_context, item_list, reserved1, reserved2*)

argument information

```
void PMDFQueueCacheGetEntry
    (PMDF_qc
     PMDF_keyword_item_list
     void
     int
     void
     int
     **cache_context,
     *item_list,
     *reserved1,
     reserved11,
     *reserved2,
     *reserved22)
```

ARGUMENTS *cache_context*
Queue cache read context created with PMDFQueueCacheGetEntry.

item_list

Item list specifying actions to be taken. **item_list** is the address of a list of item descriptors, each of which specifies an action and provides the information needed to perform that action. See the description below for further details.

reserved1, reserved11, reserved2, reserved22

These arguments are presently ignored. Pass values of zero.

DESCRIPTION PMDFqueueCacheGetEntry is used to dump the contents of the queue cache database. On the first call to the routine, the **cache_context** argument should be set to zero. It will then be created by PMDFqueueCacheGetEntry and returned along with the first queue cache entry. Repeated calls should then be made using this cache context to obtain the remaining queue cache entries. When there are no more entries to return, the context will be disposed of and PMDF__EOM returned. To prematurely abort the listing, call PMDFqueueCacheEnd.

A list of item descriptors — an item list — is used to specify, for each queue cache entry, what values to return. The **item_list** argument is the address of the first item descriptor in the list. Each item descriptor specifies an action and provides the information needed to perform that action. The list of item descriptors is terminated with an item descriptor with an `item_code` field value of `PMDF_QC_END_LIST`.

Each item descriptor has the following C-style structure declaration:

```
typedef struct {
    int    item_code;
    int    item_blength;
    void *item_address;
    int    item_length;
    int    item_status;
} PMDF_keyword_item_list;
```

where

Field name	Description
<code>item_code</code>	Item code chosen from Table 1–7 indicating the value to return. The <code>PMDF_QC_END_LIST</code> item code indicates the end of the item list. Used for input only.
<code>item_blength</code>	Maximum length in bytes of the buffer pointed at by <code>item_address</code> . For string buffers, this length does not include any null terminator. Used for input only.
<code>item_address</code>	Pointer to the buffer where the indicated value is to be written. Used for input only.
<code>item_length</code>	On output, this field is set to the length in bytes of the value written to the buffer pointed at by <code>item_address</code> . This length does not include any null terminator use to terminate string values. Used for output only.
<code>item_status</code>	Status code associated with writing the value to the buffer. Will be <code>PMDF__OK</code> for a success. In the case of an error, will generally be <code>PMDF__STRTRU</code> indicating that the value was truncated to fit. Used for output only.

The allowed item code values are given in Table 1–7. A sample program, `api_example11.pas` and `api_example12.c`, are provided in the directory of example programs, (`PMDF_ROOT:[DOC.EXAMPLES]` on OpenVMS and `/pmdf/doc/examples/` on UNIX).

PMDFqueueCacheGetEntry

Table 1-7 PMDF_queue_cache_get_entry Item Codes

Item code	Description
PMDF_QC_END_LIST	Denotes the end of the item list. The <code>item_address</code> , <code>item_blength</code> , and <code>item_length</code> fields are ignored.
PMDF_QC_CHAIN	This item entry points to another item list to process. <code>item_address</code> is a pointer to another item list to process. The <code>item_blength</code> and <code>item_length</code> fields are ignored.
PMDF_QC_CHANNEL	Name of the channel to which this message is queued. <code>item_address</code> is a pointer to a buffer of length at least <code>CHANLENGTH+1</code> bytes. The channel name is written to this buffer and null terminated.
PMDF_QC_CREATION_DATE_BIN	Binary representation of the message file's creation date and time. On OpenVMS systems, this is a quadword binary time. On UNIX systems it is a <code>time_t</code> value.
PMDF_QC_CREATION_DATE_STR	ASCII string representation of the message file's creation date and time. <code>item_address</code> should point to a buffer of length at least <code>ALFA_SIZE+1</code> bytes. The date and time will be written to that buffer and null terminated.
PMDF_QC_DEFERRED_DATE_BIN	Binary representation of any "Deferred-delivery-date:" specified in the message's RFC 822 header. Usually this value will be zero since PMDF by default ignores that header line. PMDF must be explicitly configured to honor it via the <code>deferred</code> channel keyword. On OpenVMS systems, this binary time value is a quadword binary time. On UNIX systems it is a <code>time_t</code> value.
PMDF_QC_DEFERRED_DATE_STR	ASCII string representation of any "Deferred-delivery-date:" specified in the messages RFC 822 header. <code>item_address</code> should point to a buffer of length at least <code>ALFA_SIZE+1</code> bytes. The date and time will be written to that buffer and null terminated.
PMDF_QC_EXPIRY_DATE_BIN	Binary representation of any "Expiry-date:" specified in the message's RFC 822 header. On OpenVMS systems, this binary time value is a quadword binary time. On UNIX systems it is a <code>time_t</code> value.
PMDF_QC_EXPIRY_DATE_STR	ASCII string representation of any "Expiry-date:" specified in the messages RFC 822 header. <code>item_address</code> should point to a buffer of length at least <code>ALFA_SIZE+1</code> bytes. The date and time will be written to that buffer and null terminated.
PMDF_QC_FILENAME	Full path to the message file. <code>item_address</code> should point to a buffer of length at least <code>ALFA_SIZE+1</code> bytes. The file path will be written to that buffer and null terminated.
PMDF_QC_LAST_TRY_DATE_BIN	Binary representation of the date and time at which delivery was last attempted for this message. A value of zero indicates that delivery has not yet been attempted. On OpenVMS systems, this binary time value is a quadword binary time. On UNIX systems it is a <code>time_t</code> value.
PMDF_QC_LAST_TRY_DATE_STR	ASCII string representation of the date and time at which delivery was last attempted for this message. When the message has yet to be attempted, the system's zero time representation is returned. <code>item_address</code> should point to a buffer of length at least <code>ALFA_SIZE+1</code> bytes. The date and time will be written to that buffer and null terminated.
PMDF_QC_OWNER_USERNAME	Username associated with the process which enqueued this message to PMDF. <code>item_address</code> should point to a buffer of length at least <code>ALFA_SIZE+1</code> bytes. The username will be written to that buffer and null terminated.

Table 1-7 (Cont.) PMDF_queue_cache_get_entry Item Codes

Item code	Description
PMDF_QC_PRIORITY	Processing priority assigned to the message. This is a four byte, signed integer value. Possible values are PMDF_CKEY_V_THIRD_CLASS, PMDF_CKEY_V_SECOND_CLASS, PMDF_CKEY_V_NON_URGENT, PMDF_CKEY_V_NORMAL, PMDF_CKEY_V_URGENT. <i>item_address</i> should point to the location where the value is to be written.
PMDF_QC_RECIPIENT_COUNT	Count of envelope "To:" addresses associated with the message. This is a four byte, signed integer value. <i>item_address</i> should point to the location where the value is to be written.
PMDF_QC_RECIPIENT_SYSTEM	String representation of the destination system's host name. <i>item_address</i> should point to a buffer of length at least ALFA_SIZE+1 bytes. The host name will be written to that buffer and null terminated.

RETURN VALUES

PMDF__OK	Normal, successful completion; queue cache entry returned.
PMDF__EOF	Normal, successful completion; no more queue cache entries to return.
PMDF__NO	Cannot access queue cache database. No queue cache entry returned.

PMDFreadFailureLog

Read a message delivery failure log from a message file.

PASCAL *status = PMDF_read_failure_log*
 (*dq_context, date, date_len, line, line_len*)

argument information

Argument	Data type	Access	Mechanism
dq_context	context pointer	read/write	reference
date	descriptor	read/write	reference
date_len	unsigned word	write	reference
line	descriptor	read/write	reference
line_len	unsigned word	write	reference

C *status = PMDFreadFailureLog*
 (*dq_context, date, date_len, line, line_len*)

argument information

```
int PMDFreadFailureLog(PMDF_dq **dq_context,  
                          char *date,  
                          int *date_len,  
                          char *line,  
                          int *line_len)
```

ARGUMENTS ***dq_context***
A message dequeue context created with PMDFdequeueInitialize.

date
A buffer to receive the time stamp indicating when the log record was written. Length must be at least ALFA_SIZE+1 bytes.

date_len
Length in bytes of the time stamp. Callers using PMDFreadFailureLog must, on input, supply the maximum length in bytes of **date**.

line
A buffer to receive the log line read from the message delivery failure log. Length must be at least BIGALFA_SIZE bytes.

line_len
Length in bytes of the line read. Callers using PMDFreadFailureLog must, on input, supply the maximum length in bytes of **line**.

DESCRIPTION Messages can contain a delivery failure log detailing why previous delivery attempts, if any, failed. This log can be read only after the message content (headers and body) has been read. If no log is present, then `PMDF__EOF` will be returned on the first read attempt. If however a log is present, then it can be read with repeated calls to `PMDFreadFailureLog`. After reading the last line of the log from the message, a subsequent call to `PMDFreadFailureLog` will return the `PMDF__EOF` status code. That is, if two log lines remain to be read, then the next two calls will read those two lines and return `PMDF__OK`. A third call will not read any line and will return `PMDF__EOF`.

The delivery failure log is generated with `PMDFdequeueMessageEnd` when it defers a message. It is also generated with `PMDFdeferMessage`.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. No data returned; no line read.
<code>PMDF__EOF</code>	End of message.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No data returned although a line was read.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. No data returned although a line was read.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for either date or line : descriptor for one or both has an invalid value in its <code>DSC\$B_CLASS</code> field. No data returned; however, a line was read.
<code>PMDF_NO</code>	Message read point is at the wrong location; must first read to the end of the message body with <code>PMDFreadLine</code> or <code>PMDFreadText</code> .
<code>PMDF__STRTRU</code>	Supplied string was too long; value truncated to fit.

PMDFreadHeader

PMDFreadHeader

Read a message header from a message file.

PASCAL *status* = **PMDF_read_header** (*dq_context*, *header*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>header</i>	header pointer	write	reference

C *status* = **PMDFreadHeader** (*dq_context*, *header*)

argument information

```
int PMDFreadHeader(PMDF_dq **dq_context,  
                  PMDF_hdr **header)
```

ARGUMENTS *dq_context*
A message dequeue context created with `PMDFdequeueInitialize`.

header
Address of a header structure created by `PMDFreadHeader`.

DESCRIPTION `PMDFreadHeader` will, in a single call, read the entire message header from a message. The “read point” for the message must be positioned at the start of the message header. This will be the case immediately after a call to `PMDFgetRecipient` has returned `PMDF__EOF` or after a call to `PMDFrewindMessage`.

`PMDFwriteHeader` can be called to output a header structure to a message being enqueued. `PMDFdisposeHeader` should be called to dispose of a previously read header. See Section 1.6 for details on using and manipulating header structures.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Header not read.

PMDFreadLine

Read a line from a message file.

PASCAL *status* = **PMDF_read_line** (*dq_context*, *line*, *line_len*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>line</i>	descriptor	read/write	reference
<i>line_len</i>	unsigned word	write	reference

C *status* = **PMDFreadLine** (*dq_context*, *line*, *line_len*)

argument information

```
int PMDFreadLine(PMDF_dq **dq_context,
                 char *line,
                 int *line_len)
```

ARGUMENTS

dq_context

A message dequeue context created with `PMDFdequeueInitialize`.

line

A buffer to receive the line read from the message file. Length must be at least `BIGALFA_SIZE` bytes.

line_len

Length in bytes of the line read. Callers using `PMDFreadLine` must, on input, supply the maximum length in bytes of ***line***.

DESCRIPTION

Lines from a message file can be read, one at a time, using `PMDFreadLine` or `PMDFreadText`. The only difference between `PMDFreadLine` and `PMDFreadText` is that `PMDFreadLine` removes the trailing line terminator, a line feed, from the end of the line before returning it to the caller. After reading the last line from the message, any subsequent calls to `PMDFreadLine` or `PMDFreadText` will return the `PMDF__EOF` status code. That is, if two lines remain to be read, then the next two calls will read those two lines and return `PMDF__OK`. A third call will not read any line and will return `PMDF__EOF`.

`PMDFreadLine` and `PMDFreadText` can be used to read both message header lines and the content of the message body. When either of these routines are used to read the message header, then the first blank line encountered signifies the end

PMDFreadLine

of the message header and the start of the message body. If `PMDFreadHeader` is used to read the message header, then `PMDFreadLine` and `PMDFreadText` will only read the message body and the blank line separating the message header from message body will not be seen.

`PMDFrewindMessage` can be called to reset the read position to the start of the message header.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. No data returned; no line read.
<code>PMDF__EOF</code>	End of message.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No data returned although a line was read.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. No data returned although a line was read.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for line : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. No data returned; however, a line was read.
<code>PMDF__STRTRU</code>	Supplied string was too long; value truncated to fit.

PMDFreadText

Read a line from a message file.

PASCAL *status* = **PMDF_read_text** (*dq_context*, *text*, *text_len*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>text</i>	descriptor	read/write	reference
<i>text_len</i>	unsigned word	write	reference

C *status* = **PMDFreadText** (*dq_context*, *text*, *text_len*)

argument information

```
int PMDFreadText(PMDF_dq **dq_context,
                 char *text,
                 int *text_len)
```

ARGUMENTS

dq_context

A message dequeue context created with `PMDFdequeueInitialize`.

text

A buffer to receive the line read from the message file. Length must be at least `BIGALFA_SIZE+1` bytes.

text_len

Length in bytes of the line read from the message file. Callers using `PMDFreadText` must, on input, supply the maximum length in bytes of ***text***.

DESCRIPTION

Lines from a message file can be read, one at a time, using `PMDFreadLine` or `PMDFreadText`. The only difference between `PMDFreadLine` and `PMDFreadText` is that `PMDFreadLine` removes the trailing line terminator, a line feed, from the end of the line before returning it to the caller. After reading the last line from the message, any subsequent calls to `PMDFreadLine` or `PMDFreadText` will return the `PMDF__EOF` status code. That is, if two lines remain to be read, then the next two calls will read those two lines and return `PMDF__OK`. A third call will not read any line and will return `PMDF__EOF`.

`PMDFreadLine` and `PMDFreadText` can be used to read both message header lines and the content of the message body. When either of these routines are used to read the message header, then the first blank line encountered signifies the end

PMDFreadText

of the message header and the start of the message body. If `PMDFreadHeader` is used to read the message header, then `PMDFreadLine` and `PMDFreadText` will only read the message body and the blank line separating the message header from message body will not be seen.

`PMDFrewindMessage` can be called to reset the read position to the start of the message header.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. No data returned; no line read.
<code>PMDF__EOF</code>	End of message.
<code>PMDF__FATERRLIB</code>	Call to <code>LIB\$SCOPY_R_DX</code> failed owing to a fatal internal error in the OpenVMS Run Time Library. No data returned although a line was read.
<code>PMDF__INSVIRMEM</code>	Insufficient virtual memory: call to <code>LIB\$GET_VM</code> made by <code>LIB\$SCOPY_R_DX</code> has failed. No data returned although a line was read.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for text : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. No data returned; however, a line was read.
<code>PMDF__STRTRU</code>	Supplied string was too long; value truncated to fit.

PMDFreceiptControl

Control the generation of read and delivery receipts.

PASCAL *status = PMDF_receipt_control*
 (*nq_context, read, delivery, read_comment,*
 delivery_comment, suppress_receipts)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference
read	signed longword	read	value
delivery	signed longword	read	value
read_comment	boolean	read	value
delivery_comment	boolean	read	value
suppress_receipts	boolean	read	value

C *status = PMDFreceiptControl*
 (*nq_context, read, delivery, read_comment,*
 delivery_comment, suppress_receipts)

argument information

```
int PMDFreceiptControl(PMDF_nq **nq_context,
                      int read,
                      int delivery,
                      int read_comment,
                      int delivery_comment,
                      int suppress_receipts)
```

ARGUMENTS

nq_context
 A message enqueue context created with PMDFstartMessageEnvelope.

read
 The value -1, 0, or +1. See the Description for details.

delivery
 The value -1, 0, or +1. See the Description for details.

read_comment
 If true then read receipt request comments will be honored; otherwise, read receipt request comments will be ignored.

PMDFreceiptControl

delivery_comment

If true then delivery receipt request comments will be honored; otherwise, delivery receipt request comments will be ignored.

suppress_receipts

If true then any read or delivery receipt request headers will be removed from a message's header prior to enqueueing it. If false, then read and delivery receipt headers will not be removed if present.

DESCRIPTION

PMDFreceiptControl can be called to set or alter the nature of the read and delivery receipt headers which PMDF can generate. The settings established by PMDFreceiptControl will only affect the specified message enqueue context and can be changed with further calls to PMDFreceiptControl.

By calling PMDFreceiptControl prior to each call to PMDFaddRecipient, the receipt handling behavior can be altered on a per address basis. It is important to keep in mind that when a message with multiple recipients is enqueued, multiple copies of that message can actually be created. Each copy differing in the contents of the message envelope and message header. In this way, it is possible to enqueue a message which will have receipt requests for some addressees but not others. A copy is made for those addressees requiring read receipt requests, another copy for those requiring delivery receipt requests, a third for those requiring both, and another for those requiring neither. Actually, it is even more complicated than this as different receipt request addresses can appear.

The **read** and **delivery** arguments have default values of 0. These two arguments set the default receipt generation behavior:

- 1. By default, if no other mechanism causes the creation of a read [delivery] receipt request, then an explicit "Read-receipt-to: <>" ["Delivery-receipt-to: <>"] header line is added to the message header. This has the effect of blocking any read [delivery] receipts from being returned to the message's originator.
0. By default, no read [delivery] receipt request headers are added to the message header.
1. By default, a read [delivery] receipt request header is added to the message header. The return address used for the header is that of the message's originator (envelope "From:" address) unless some other address has been selected with PMDFsetReceiptAddresses.

The **read_comment** and **delivery_comment** arguments control whether or not comment strings in "To:", "Cc:", and "Bcc:" addresses can be used to request a read or delivery receipt from that particular addressee. By default, such comments are ignored. To honor comments requesting read [delivery] receipts, specify a true value for **read_comment** [**delivery_comment**]; to ignore comments requesting read [delivery] receipts, specify a false value for **read_comment** [**delivery_comment**]. See the discussion of read and delivery receipt requests in the *PMDF System Manager's Guide* for further details on the use of comment strings in addresses as receipt requests.

Finally, the **suppress_receipts** argument can be used to forcibly strip any or all receipt requests from a message's header. If **suppress_receipts** is true, then this

stripping will always be done and will override any other mechanism for specifying receipt requests. If **suppress_receipts** is false, then such blind stripping will not be performed and the other mechanisms will be allowed to function. This is the default case.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. No settings made or changed.

PMDFrecipientDisposition

PMDFrecipientDisposition

Specify the disposition of a dequeued recipient address.

PASCAL *status* = **PMDF_recipient_disposition**
 (*dq_context*, *notary_flags*, *disposition*, *address*,
 orig_address, *reason*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>notary_flags</i>	integer	read	value
<i>disposition</i>	integer	read	value
<i>address</i>	descriptor	read	reference
<i>orig_address</i>	descriptor	read	reference
<i>reason</i>	descriptor	read	reference

C *status* = **PMDFrecipientDisposition**
 (*dq_context*, *notary_flags*, *disposition*, *address*,
 address_len, *orig_address*, *orig_address_len*, *reason*,
 reason_len)

argument information

```
int PMDFrecipientDisposition(PMDF_dq      **dq_context,  
                             int         notary_flags,  
                             int         disposition,  
                             char        *address,  
                             int         address_len,  
                             char        *orig_address,  
                             int         orig_address_len,  
                             char        *reason,  
                             int         reason_len)
```

ARGUMENTS

dq_context

A message dequeue context created with `PMDFdequeueInitialize`.

notary_flags

NOTARY flags for this envelope recipient address as obtained from a prior call to `PMDFgetRecipientFlags`.

disposition

Disposition for this envelope recipient address.

address

Envelope recipient address obtained from `PMDFgetRecipient` and being reported on.

address_len

Length in bytes of the envelope recipient address.

orig_address

Original form of the envelope recipient address obtained from `PMDFgetRecipient` and being reported on.

orig_address_len

Length in bytes of the original envelope recipient address.

reason

Optional text string describing the disposition of the envelope recipient address being reported on. The length of this string should not exceed `BIGALFA_SIZE` bytes.

reason_len

Length in bytes of **reason**.

DESCRIPTION As part of message dequeue processing, a list of envelope recipient addresses is obtained by repeatedly calling `PMDFgetRecipient`. Once the disposition of each envelope recipient address is known (*e.g.*, delivered, failed, relayed, deferred, *etc.*), that disposition should be conveyed back to PMDF. When the processing of the message is completed, PMDF can automatically determine how to dispose of the message, as described below. See the description of `PMDFdequeueMessageEnd` for further details.

The value of **notary_flags** should be the value obtained from `PMDFgetRecipientFlags`. The value of **disposition** must be chosen from Table 1–8 and states the disposition of the envelope recipient address being reported on.

Table 1–8 Disposition Values for Use with `PMDF_recipient_disposition`

Symbolic name	Value	Description
<code>PMDF_DISP_DEFERRED</code>	1	Recipient address processing failed owing to a temporary problem (<i>e.g.</i> , network down, remote host unreachable, mailbox busy, <i>etc.</i>); defer processing of this address until later.
<code>PMDF_DISP_DELIVERED</code>	2	Recipient address successfully delivered; generate a delivery status notification if required.
<code>PMDF_DISP_FAILED</code>	3	Recipient address processing has failed owing to a permanent problem (<i>e.g.</i> , invalid recipient address, recipient over quota, <i>etc.</i>); no further delivery attempts should be made for this address; generate a non-delivery notification if required.
<code>PMDF_DISP_RELAYED</code>	4	Recipient address forwarded to another address or gatewayed into a non-NOTARY mail system; the message's NOTARY information was, however, preserved; there is no need to generate a "relayed" notification message.

PMDFrecipientDisposition

Table 1–8 (Cont.) Disposition Values for Use with PMDF_recipient_disposition

Symbolic name	Value	Description
PMDF_DISP_RELAYED_FOREIGN	5	Recipient address forwarded to another address or gatewayed to a non-NOTARY mail system; the message's NOTARY information was not preseved; generate a "relayed" notification message if required.
PMDF_DISP_RETURN	6	For this recipient, return the message as undeliverable; generate a non-delivery notification if required.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	Illegal value specified for disposition . Disposition not set.
PMDF__BADCONTEXT	Illegal or corrupt context, dq_context . Disposition not set.
PMDF__INVSTRDES	Invalid string descriptor for address , orig_address , or reason : one or more of the descriptors has an invalid value in its DSC\$B_CLASS field. Disposition not set.
PMDF__STRTRUERR	One or both of the address or orig_address strings exceeds the maximum permitted length. Disposition not set.

PMDFReturnMessage

Return a message to its originator.

PASCAL *status* = **PMDF_return_message**
 (*dq_context*, *channel*, *from*, *bad_addresses*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference
<i>channel</i>	descriptor	read	reference
<i>from</i>	descriptor	read	reference
<i>bad_addresses</i>	item list	read	reference

C *status* = **PMDFReturnMessage**
 (*dq_context*, *channel*, *channel_len*, *from*, *from_len*,
 bad_addresses)

argument information

```
int PMDFReturnMessage(PMDF_dq           **dq_context,
                      char               *channel,
                      int                channel_len,
                      char               *from,
                      int                from_len,
                      PMDF_item_list   *bad_addresses)
```

ARGUMENTS

dq_context

A message dequeue context created with `PMDFdequeueInitialize`.

channel

Name of the channel to act in behalf of when bouncing the message. The length of ***channel*** must not exceed `CHANLENGTH` bytes.

channel_len

Length in bytes of ***channel***.

from

Envelope "From:" address associated with the message to be returned. This string was returned by `PMDFgetMessage` and must not exceed `ALFA_SIZE` bytes.

from_len

Length in bytes of the envelope "From:" address. This value was returned by `PMDFgetMessage`.

PMDFreturnMessage

bad_addresses

Item list specifying each bad address along with any error information.

DESCRIPTION NOTE: While still supported, this routine is now obsolete. Callers should instead use the `PMDFrecipientDisposition` routine to stipulate the disposition of each recipient address. Then, when `PMDFdequeueMessageEnd` or `PMDFdequeueMessage` is called, any necessary notification messages will automatically be generated. Moreover, the notification messages will conform to the NOTARY specifications (RFC 1892, 1893, and 1894).

NOTE: The notification messages generated by `PMDFreturnMessage` do not adhere to the NOTARY specifications.

Messages can be returned to their originator with `PMDFreturnMessage`. Messages will be returned in behalf of the channel specified. If no channel name is specified (**channel** has zero length), then `PMDFreturnMessage` will use the name of the currently running channel if possible and the local channel otherwise. In order to remove the returned message from PMDF's message queues, `PMDFdequeueMessageEnd` should be called after calling `PMDFreturnMessage`.

`PMDFreturnMessage` will determine from the message's header the most appropriate address to return the message to as well as whether or not to send a copy of the message to the local postmaster (as controlled by channel keywords for the channel the message is being returned in whose behalf).

The returned message will be a multipart message containing two parts. The first part contains a list of the bad addresses to which the original message was addressed to. These addresses are given in the item list referenced by **bad_addresses**. Specifically, the **bad_addresses** argument is the address of a list of item descriptors, each of which describes a bad address. Each item descriptor has the structure

```
struct {
    int    reserved1;
    void *item_address;
    int    reserved2;
    int    item_length;
}
```

`item_address` is a pointer to a string giving a bad address and any explanation as to why the address was bad; `item_length` is an integer giving the length of the string pointed at by `item_address`. The item list is terminated by an entry with an `item_length` of zero.

Each string specified by an entry in the item list is output, one string per line. The strings appear best if in the format:

```
address - error text
```

where *address* is a bad address and *error-text* is any applicable error message associated with the bad address. The bad addresses are generally envelope "To:" addresses which failed. For example,

a@b.com - mail rejected; no such user 'a@b.com'.

The second part of the multipart message will contain the failed message itself.

Examples 1–10 and 1–11 demonstrate the use of `PMDfreturnMessage`.

RETURN VALUES

<code>PMDf__OK</code>	Normal, successful completion.
<code>PMDf__BADCONTEXT</code>	Illegal or corrupt context. Message not returned.
<code>PMDf__INVSTRDES</code>	Invalid string descriptor for channel or from : one or both of the descriptors has an invalid value in its <code>DSC\$B_CLASS</code> field. Message not returned.
<code>PMDf__STRTRUERR</code>	One or both of the channel or from strings exceeds the maximum permitted length. Message not returned.

PMDFrewindMessage

PMDFrewindMessage

Rewind a message file back to the start of its message header.

PASCAL *status* = **PMDF_rewind_message** (*dq_context*)

argument information

Argument	Data type	Access	Mechanism
<i>dq_context</i>	context pointer	read/write	reference

C *status* = **PMDFrewindMessage** (*dq_context*)

argument information

```
int PMDFrewindMessage(PMDF_dq **dq_context)
```

ARGUMENTS *dq_context*
A message dequeue context created with `PMDFdequeueInitialize`.

DESCRIPTION `PMDFrewindMessage` will “rewind” a message file back to the start of its message header. This routine can be called any time after all of the envelope "To:" addresses have been read with `PMDFgetRecipient` and prior to dequeuing or deferring the message. After `PMDFrewindMessage` has been called, the message header can be read with either `PMDFreadHeader`, `PMDFreadLine`, or `PMDFreadText`.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Message was not rewound.
<code>PMDF__NO</code>	There is some sort of inconsistency in the message file; the message cannot be rewound.

PMDFsetCallBack

Specify the address of a procedure to call when a PMDF RESTART or PMDF SHUTDOWN command has been issued.

PASCAL *status* = **PMDF_set_call_back**
 (*proc*, *facility*, *facility_len*)

argument information

Argument	Data type	Access	Mechanism
<i>proc</i>	procedure	read	reference
<i>facility</i>	descriptor	read	reference
<i>facility_len</i>	signed longword	read	value

C *status* = **PMDFsetCallBack**
 (*proc*, *facility*, *facility_len*)

argument information

```
int PMDFsetCallBack(void (*proc)(),
                    char *facility,
                    int  facility_len)
```

ARGUMENTS

proc

An asynchronous procedure which will be called at AST level whenever a PMDF RESTART or SHUTDOWN command is issued. This procedure will be passed by reference a single integer parameter explaining the reason for the call back.

facility

Facility or component name to associate with the routine using this call back. When a RESTART or SHUTDOWN command specifying this facility name is issued, then the call back procedure will be invoked. The length of this string should not exceed 17 bytes.

facility_len

Length in bytes of ***facility***.

PMDFsetCallBack

DESCRIPTION PMDFsetCallBack is only functional on OpenVMS systems. On other systems, it merely returns PMDF__OK and does nothing.)

Through a call back procedure, programs can be notified whenever a PMDF RESTART or PMDF SHUTDOWN command has been issued. Unless PMDF-cancelCallBack is called, the call back procedure will be called each and every time any of the five commands are issued

```
$ PMDF CACHE/CLOSE
$ PMDF RESTART
$ PMDF RESTART facility
$ PMDF SHUTDOWN
$ PMDF SHUTDOWN facility
```

where *facility* is the facility name.

The call back procedure will be invoked at AST level and passed, by reference, a single argument. This argument is of type integer and has one of three values:

Symbolic name	Value	Command issued
PMDF_CACHE_CALLBACK	8	PMDF CACHE/CLOSE
PMDF_RESTART_CALLBACK	16	PMDF RESTART [facility]
PMDF_SHUTDOWN_CALLBACK	24	PMDF SHUTDOWN [facility]

In response to a PMDF_CACHE_CALLBACK call back, the program using the call back should close the queue cache as soon as is convenient by calling PMDFcloseQueueCache. In response to either of the other two call backs, the program should exit in an orderly fashion as soon as is convenient. In the case of PMDF_RESTART_CALLBACK, the program should be restarted (*i.e.*, re-run).

On OpenVMS systems, this routine will enqueue five resource locks each with blocking ASTs. In order to accomplish this, SYSLCK privilege as well as a sufficient enqueue and AST quotas are required. The call back procedure will be invoked at AST level. Note that the delivery of the blocking AST's used by PMDFsetCallBacks can be hindered in a program which itself spends most of its time at AST level.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__INVSTRDES	Invalid string descriptor for facility : descriptor has an invalid value in its DSCB_CLASS field. No call back established.
PMDF__STRTRUERR	Length of facility exceeds 17 bytes. No call back established.
On OpenVMS systems	Any error returned by the \$ENQ System Service.

PMDFsetEnvelopeId

Specify the envelope id to associate with this message.

PASCAL *status = PMDF_set_envelope_id*
 (*nq_context, envelope_id*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>envelope_id</i>	descriptor	read	reference

C *status = PMDFsetEnvelopeId*
 (*nq_context, envelope_id, envelope_id_len*)

argument information

```
int PMDFsetEnvelopeId(PMDF_nq **nq_context,
                      char *envelope,
                      int envelope_id_len)
```

ARGUMENTS

nq_context

A message enqueue context created with `PMDFstartMessageEnvelope`.

envelope_id

Envelope id to use for this message. Length can not exceed `ALFA_SIZE` bytes.

envelope_id_len

Length in bytes of the envelope id.

DESCRIPTION

Messages queued to PMDF carry with them two identification strings – “id’s” for short. The first is the “message id” as seen in the message’s RFC 822 “Message-id:” header line. This id is the same for all copies of a given message. The second id is the envelope id. Each copy of the message has a distinct envelope id.

Normally you will only specify these id’s yourself when you are re-enqueuing a message to PMDF. In that case, it is important to preserve the envelope id and message id. If you are enqueueing a new message to PMDF, then you should just

PMDFsetEnvelopeId

leave generation of these id's to PMDF: PMDF will automatically generate both of these id's when they are not supplied.

Should you want to set the message id, then include your own Message-id: header line in the enqueued message's RFC 822 header. If you want to set the envelope id, then do so with this routine. Note, however, that if PMDF has to make multiple copies of the enqueued message, then it is likely that your specified envelope id will not be used. Your message id, however, will be used since a message id is identical across all copies of the message.

When re-enqueuing a dequeued message to PMDF, you can get obtain the envelope id and NOTARY flags of the dequeued message via the `PMDFgetEnvelopeId` and `PMDFgetRecipientFlags` routines. You would then propogate the id and flags forward by calling `PMDFsetEnvelopeId` once after `PMDFstartMessageEnvelope`, and by calling `PMDFsetRecipientFlags` once for each, and prior to each, `PMDFaddRecipient` call.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. No envelope id set.
<code>PMDF__INVSTRDES</code>	Invalid string descriptor for envelope_id : descriptor has an invalid value in its <code>DSC\$B_CLASS</code> field. Envelope id not set.
<code>PMDF__STRTRUERR</code>	Supplied string was too long. Envelope id not set.

PMDFsetLimits

Set message fragmentation thresholds.

PASCAL *status = PMDF_set_limits*
 (*nq_context, max_blocks, max_lines, max_to*)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference
max_blocks	signed longword	read	value
max_lines	signed longword	read	value
max_to	signed longword	read	value

C *status = PMDFsetLimits*
 (*nq_context, max_blocks, max_lines, max_to*)

argument information

```
int PMDFsetLimits(PMDF_nq **nq_context,
                 int         max_blocks,
                 int         max_lines,
                 int         max_to)
```

ARGUMENTS

nq_context

A message enqueue context created with PMDFstartMessageEnvelope.

max_blocks

Non-negative integer specifying the maximum number of blocks (header + body) per message. A value of zero implies no limit.

max_lines

Non-negative integer specifying the maximum number of message lines (header + body) per message. A value of zero implies no limit.

max_to

Non-negative integer specifying the maximum number of envelope "To:" addresses per message. A value of zero implies no limit.

PMDFsetLimits

DESCRIPTION PMDF can be instructed to fragment “large” messages into multiple messages. Large is taken by PMDF to mean exceeds **max_blocks** blocks, exceeds **max_lines** message lines, or exceeds **max_to** envelope "To:" addresses. All of these limits are simultaneously imposed. When either **max_blocks** or **max_lines** is exceeded, the message is fragmented into multiple messages using MIME’s message/partial mechanism. MIME compliant mailers receiving the message can automatically re-assemble the message upon receipt of all of the pieces. (PMDF channels must be marked with the `defragment` keyword for automatic message re-assembly to occur.) When the **max_to** limit is exceeded, the message is merely broken into multiple copies, each copy with an envelope "To:" address list of length less than or equal to **max_to**.

Note that the size of a block in bytes is given by the PMDF option file entry `BLOCK_SIZE`. When not specified in an option file, the default value of 1024 bytes is used. The function `PMDFgetBlockSize` should be used to determine the current block size.

Settings chosen with `PMDFsetLimits` only affect the specified message enqueue context and can be changed with further calls to `PMDFsetLimits`. By default, no limits are imposed: **max_blocks** = **max_lines** = **max_to** = 0.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Limits were not changed.

PMDFsetMutex

Provide mutex handling routines.

PASCAL *status* = **PMDF_set_mutex**
 (*create, lock, unlock, delete, sleep*)

argument information

Argument	Data type	Access	Mechanism
create	procedure	read	reference
lock	procedure	read	reference
unlock	procedure	read	reference
delete	procedure	read	reference
sleep	procedure	read	reference

C *status* = **PMDFsetMutex**
 (*create, lock, unlock, delete, sleep*)

argument information

```
int PMDFsetMutex(int (*create)(),
                 int (*lock)(),
                 int (*unlock)(),
                 int (*delete)(),
                 void (*sleep)())
```

ARGUMENTS

create

Address of a procedure to create a mutex.

lock

Address of a procedure to lock a mutex.

unlock

Address of a procedure to unlock a mutex.

delete

Address of a procedure to delete a mutex.

sleep

Address of a procedure to sleep the specified number of hundredths of a second.

PMDFsetMutex

DESCRIPTION The PMDF API and underlying routines are re-entrant and thread-safe. Multi-threaded routines which will be using the PMDF API must call `PMDFsetMutex` before calling any other API routines, including `PMDFinitialize`. The procedures passed to `PMDFsetMutex` are then used by PMDF to manage thread mutexes and efficiently sleep a thread.

The procedures referenced by **create**, **lock**, **unlock**, and **delete** each perform the mutex operation implied by their name:

create: Create and initialize a mutex.

lock: Block other threads wanting to use the mutex.

unlock: Allow other threads to use the mutex.

delete: Destroy the mutex and free up any memory associated with it.

Each of the four routines accept a single parameter which is the address of a pointer to a thread mutex. That is, if a thread mutex is the structure `MUTEX` then the routines would be declared in C as

```
int create (struct MUTEX **mutex)
int lock (struct MUTEX **mutex)
int unlock (struct MUTEX **mutex)
int delete (struct MUTEX **mutex)
```

The mutex creation routine should create the mutex, initialize it, and return the address of the mutex. The integer return value should be 0. It is not presently used by PMDF, but is provided for compatibility with POSIX Threads mutex routines. For example,

```
int create (struct MUTEX **mutex)
{
    struct MUTEX *mtx;

    mtx = (struct MUTEX *)calloc (sizeof (struct MUTEX));
    mutex_init (mtx);
    *mutex = mtx;
    return (0);
}
```

Routines must not assume that only one mutex will be used by PMDF. PMDF creates and uses a number of mutexes.

The procedure referenced by **sleep** accepts an unsigned longword passed by value and specifying the number of hundredths of seconds to sleep:

```
void sleep (unsigned long centi_seconds)
```

The sleep procedure is not expected to return a value. Optionally, a value of zero can be supplied for **sleep** in which case PMDF will use a simple, non-thread aware routine to sleep the process.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BAD	One or more of the parameters create , lock , unlock , or delete was zero. Mutex routines not set.
PMDF__NO	PMDFinitialize was called prior to PMDFsetMutex; this should be treated as a fatal error.

PMDFsetRecipientFlags

PMDFsetRecipientFlags

Set the NOTARY flags for the next envelope recipient address.

PASCAL *status* = **PMDF_set_recipient_flags**
 (*nq_context*, *notary_flags*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>notary_flags</i>	integer	read	value

C *status* = **PMDFsetRecipientFlags**
 (*nq_context*, *notary_flags*)

argument information

```
int PMDFsetRecipientFlags(PMDF_nq **dq_context,  
int notary_flags)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

notary_flags
Longword integer containing NOTARY flag bits.

DESCRIPTION PMDF mail messages carry per recipient NOTARY information in their envelope. This information is aligned with the NOTARY SMTP extension as described in RFC 1891 and describes failure and success handling requested by the sender (*e.g.*, send a delivery receipt, send failure notifications but do not include return of content, never send any form of notifications, *etc.*).

By default, when an envelope recipient address is enqueued, PMDF assigns it the NOTARY handling `PMDF_RECEIPT_FAILURES + PMDF_RECEIPT_DELAYS` which indicates that non-delivery notifications (NDNs) should be generated for delivery failures and delays. To select, for a given envelope recipient address, different handling characteristics or to propagate NOTARY flags from a previous dequeue operation, call `PMDFsetRecipientFlags` prior to calling

PMDFaddRecipient. The **notary_flags** argument is a bit mask whose bits are given in Table 1-6.

Note that PMDF_RECEIPT_NEVER and PMDF_RECEIPT_FAILURES can not both be set. If both are set, then PMDF_RECEIPT_NEVER will be ignored. Similarly, if both PMDF_RECEIPT_HEADER and PMDF_RECEIPT_NOHEADER are set, then PMDF_RECEIPT_NOHEADER is ignored. When neither are set, then notifications will include full return of content (RET=FULL).

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. No flags set.

PMDFsetRecipientType

PMDFsetRecipientType

Specify whether subsequent addresses are To:, Cc:, or Bcc: addresses.

PASCAL *status* = **PMDF_set_recipient_type**
 (*nq_context*, *to*, *cc*, *bcc*, *envelope*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>to</i>	boolean	read	value
<i>cc</i>	boolean	read	value
<i>bcc</i>	boolean	read	value
<i>envelope</i>	boolean	read	value

C *status* = **PMDFsetRecipientType**
 (*nq_context*, *to*, *cc*, *bcc*, *envelope*)

argument information

```
int PMDFsetRecipientType(PMDF_nq **nq_context,  
                          int to,  
                          int cc,  
                          int bcc,  
                          int envelope)
```

ARGUMENTS

nq_context

A message enqueue context created with `PMDFstartMessageEnvelope`.

to

If true, then subsequent addresses added with `PMDFaddRecipient` will be treated as "To:" addresses (and possibly as "Cc:" or "Bcc:" addresses too). If false, then subsequent addresses will not be treated as "To:" addresses.

cc

If true, then subsequent addresses added with `PMDFaddRecipient` will be treated as "Cc:" addresses (and possibly as "To:" or "Bcc:" addresses too). If false, then subsequent addresses will not be treated as "Cc:" addresses.

bcc

If true, then subsequent addresses added with `PMDFaddRecipient` will be treated as "Bcc:" addresses (and possibly as "To:" or "Cc:" addresses too). If false, then subsequent addresses will not be treated as "Bcc:" addresses.

envelope

If true, then all subsequent addresses added with `PMDFaddRecipient` will be added to the message envelope as envelope "To:" addresses. If false, then subsequent addresses will not be added to the message envelope but can be added to the message header lines.

DESCRIPTION When `PMDFstartMessageEnvelope` is called, the defaults **to** = true, **cc** = false, **bcc** = false, **envelope** = true are established. These defaults can then be changed by calls to `PMDFsetRecipientType` which can be called as often as is necessary while building the message envelope with `PMDFaddRecipient` calls. Note that any combination of **to**, **cc**, or **bcc** can simultaneously be set true. For instance, if **to** and **cc** are set true, then any address added with `PMDFaddRecipient` will be treated as both a "To:" and "Cc:" address. It will be added only once to the message envelope if **envelope** is true, but will appear in both the "To:" and "Cc:" message header line.

The settings made with `PMDFsetRecipientType` only affect the specified message enqueue context and can be subsequently altered by subsequent calls to `PMDFsetRecipientType`.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Recipient type not changed.

PMDFsetReceiptAddresses

Specify delivery and read receipt request addresses for a message being enqueued.

PASCAL *status* = **PMDF_set_receipt_addresses**
 (*nq_context*, *read_address*, *delivery_address*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>read_address</i>	descriptor	read	reference
<i>delivery_address</i>	descriptor	read	reference

C *status* = **PMDFsetReceiptAddresses**
 (*nq_context*, *read_address*, *read_address_len*,
 delivery_address, *delivery_address_len*)

argument information

```
int PMDFsetReceiptAddresses(PMDF_nq **nq_context,  
                             char      *read_address,  
                             int       read_address_len,  
                             char      *delivery_address,  
                             int       delivery_address_len)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

read_address
Address to send a read receipt to. Length can not exceed `ALFA_SIZE` bytes.

read_address_len
Length in bytes of *read_address*.

delivery_address
Address to send a delivery receipt to. Length can not exceed `ALFA_SIZE` bytes.

delivery_address_len
Length in bytes of *delivery_address*.

DESCRIPTION PMDFsetReceiptAddresses can be called to set default values for the addresses to which to send read or delivery receipts. If either string has zero length, then no default will be set for the associated receipt address. These addresses will then be used in the construction of read or delivery receipt request header lines whenever a read or delivery receipt is requested for the specified message enqueue context. Note that these default addresses can be overridden by other receipt request mechanisms or suppressed in response to PMDFreceiptControl call with **suppress_receipts** set true.

By default, no read or delivery receipt addresses are set. Settings made with this routine only affect the specified message enqueue context and can be further changed by additional calls to PMDFsetReceiptAddresses.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. Receipt addresses not changed.
PMDF__INVSTRDES	Invalid string descriptor for read_address or delivery_address : one or both of the descriptors has an invalid value in its DSC\$B_CLASS field. Receipt addresses not changed.
PMDF__STRTRUERR	One or both of the input strings exceeds ALFA_SIZE bytes. Receipt addresses not changed.

PMDFstartMessageBody

PMDFstartMessageBody

Begin the body of a message which is being enqueued.

PASCAL *status* = **PMDF_start_message_body** (*nq_context*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference

C *status* = **PMDFstartMessageBody** (*nq_context*)

argument information

```
int PMDFstartMessageBody(PMDF_nq **nq_context)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

DESCRIPTION After the message header has been written, `PMDFstartMessageBody` should be called to begin the message body. If the message has no body, then `PMDFenqueueMessage` should be called without calling `PMDFstartMessageBody`.

After `PMDFstartMessageBody` has been called, either `PMDFwriteLine` or `PMDFwriteText` must be used to write the message body. Once the message body is complete, `PMDFenqueueMessage` should be used to enqueue the message.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Message body not started.

PMDFstartMessageEnvelope

Begin a message enqueue; specify the envelope "From:" address.

PASCAL *status* = **PMDF_start_message_envelope**
 (*nq_context*, *channel*, *from*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	write	reference
<i>channel</i>	descriptor	read	reference
<i>from</i>	descriptor	read	reference

C *status* = **PMDFstartMessageEnvelope**
 (*nq_context*, *channel*, *channel_len*, *from*, *from_len*)

argument information

```
int PMDFstartMessageEnvelope(PMDF_nq **nq_context,
                             char      *channel,
                             int       channel_len,
                             char      *from,
                             int       from_len)
```

ARGUMENTS

nq_context
 Message enqueue context created for this message enqueue context.

channel
 Name of the channel to act as when enqueueing the message. Length can not exceed CHANLENGTH bytes.

channel_len
 Length in bytes of ***channel***.

from
 Envelope "From:" address for the message to be enqueued. Length can not exceed ALFA_SIZE bytes.

from_len
 Length in bytes of the envelope "From:" address.

PMDFstartMessageEnvelope

DESCRIPTION PMDFstartMessageEnvelope must be called to start a message enqueue context. No other message enqueue API procedures can be called until after PMDFstartMessageEnvelope has been called.

For programs which act as a user interface, the local channel name, "1", should be used for the **channel** argument. Channel programs should use their own channel name. If a zero length string is passed in, then "1" will be used if the **ischannel** argument of PMDFinitialize was false; otherwise, PMDFgetChannelName will be called to determine the current channel name and that will be used.

The **from** argument specifies the envelope "From:" address to associate with the message to be enqueued. An envelope "From:" address must be specified and should conform to RFC 822. PMDF will do its best to transform non-conformant addresses into legal RFC 822 addresses; however, this is not always possible and a PMDF__NO error can result.

After calling PMDFstartMessageEnvelope, PMDFaddRecipient should be called to specify all "To:", "Cc:", and "Bcc:" addresses.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__INVSTRDES	Invalid string descriptor for channel or from : one or both of the descriptors has an invalid value in its DSC\$B_CLASS field. Message enqueue context not started.
PMDF__NO	Error initializing PMDF. Either the specified channel does not exist or a problem exists with the site's PMDF configuration (e.g., duplicate channel name in the configuration file). PMDFgetErrorText can be called to obtain additional information about the nature of the error.
PMDF__STRTRUERR	One or both of the input strings is too long. Message enqueue context not started.

PMDFstartMessageHeader

Begin the message header of a message which is being enqueued.

PASCAL *status* = **PMDF_start_message_header** (*nq_context*)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference

C *status* = **PMDFstartMessageHeader** (*nq_context*)

argument information

```
int PMDFstartMessageHeader(PMDF_nq **nq_context)
```

ARGUMENTS

nq_context

A message enqueue context created with PMDFstartMessageEnvelope.

DESCRIPTION

After the message envelope has been constructed by calls to PMDFaddRecipient, the construction of the message header is started with a call to PMDFstartMessageHeader. Header lines can be written with PMDFwriteHeader, PMDFwriteLine, PMDFwriteText, PMDFwriteFrom, PMDFwriteDate, and PMDFwriteSubject. The only mandatory header lines which must be written are the "From:" and "Date:" header lines. PMDF will supply all other required header lines.

RETURN VALUES

PMDF__OK

Normal, successful completion.

PMDF__BADCONTEXT

Illegal or corrupt context. Message header not started.

PMDFwriteDate

PMDFwriteDate

Write a "Date:" header line to a message being enqueued.

PASCAL *status* = **PMDF_write_date** (*nq_context*)

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference

C *status* = **PMDFwriteDate** (*nq_context*)

argument information

```
int PMDFwriteDate(PMDF_nq **nq_context)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

DESCRIPTION `PMDFwriteDate` will output a "Date:" header line to a message header. The current system date and time will be used in constructing this header line.

`PMDFwriteDate` should be called after `PMDFstartMessageHeader` and prior to calling `PMDFstartMessageBody`. If it is called after `PMDFstartMessageBody`, then its output will become part of the message body.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. "Date:" header line not written.

PMDFwriteFrom

Write a "From:" header line to a message being enqueued.

PASCAL *status* = **PMDF_write_from** (*nq_context*, *from*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>from</i>	descriptor	read	reference

C *status* = **PMDFwriteFrom**
 (*nq_context*, *from*, *from_len*)

argument information

```
int PMDFwriteFrom(PMDF_nq **nq_context,
                  char      *from,
                  int       from_len)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

from
Envelope "From:" address for the message to be enqueued. Length can not exceed `ALFA_SIZE` bytes.

from_len
Length in bytes of the envelope "From:" address.

DESCRIPTION `PMDFwriteFrom` will output a "From:" header line to a message header. The address cited in the header line will be that supplied with the **from** argument.

`PMDFwriteFrom` should be called after `PMDFstartMessageHeader` and prior to calling `PMDFstartMessageBody`. If it is called after `PMDFstartMessageBody`, then it's output will become part of the message body.

PMDFwriteFrom

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. "Date:" header line not written.
PMDF__INVSTRDES	Invalid string descriptor for from : descriptor has an invalid value in its DSC\$B_CLASS field. Header line not written.
PMDF__STRTRUERR	The from input string is too long. Header line not written.

PMDFwriteHeader

Write a message header to a message being enqueued.

PASCAL *status* = **PMDF_write_header** (*nq_context*, *header*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>header</i>	header pointer	read	value

C *status* = **PMDFwriteHeader** (*nq_context*, *header*)

argument information

```
int PMDFwriteHeader(PMDF_nq **nq_context,
                   PMDF_hdr *header)
```

ARGUMENTS *nq_context*
A message enqueue context created with `PMDFstartMessageEnvelope`.

header
Address of a message header structure created with `PMDFreadHeader` or `PMDFaddHeaderLine`.

DESCRIPTION Header structures can be output with `PMDFwriteHeader`. See Section 1.6 for details on using and manipulating header structures.

RETURN VALUES

<code>PMDF__OK</code>	Normal, successful completion.
<code>PMDF__BADCONTEXT</code>	Illegal or corrupt context. Header not written.

PMDFwriteLine

Write a line of text to a message being enqueued.

PASCAL *status* = **PMDF_write_line** (*nq_context*, *line*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>line</i>	descriptor	read	reference

C *status* = **PMDFwriteLine** (*nq_context*, *line*, *line_len*)

argument information

```
int PMDFwriteLine(PMDF_nq **nq_context,  
                  char      *line,  
                  int       line_len)
```

ARGUMENTS

nq_context

A message enqueue context created with `PMDFstartMessageEnvelope`.

line

Line of text to write to the message. Length can not exceed 65,535 bytes.

line_len

Length in bytes of ***line***.

DESCRIPTION

Text can be written to a message using `PMDFwriteLine` or `PMDFwriteText`. The only difference between these two routines is that `PMDFwriteLine` always appends a record terminator, line feed, to the end of each line it outputs. `PMDFwriteText` does not: it is left to callers of `PMDFwriteText` to include record terminators, where appropriate, in their output.

Each line written with `PMDFwriteLine` will appear as a single line (record) in the message being composed. For this reason, `PMDFwriteLine` is often more convenient to use than `PMDFwriteText`. However, programs which loop reading lines from a queued message and writing them to a new message should use `PMDFreadText` and `PMDFwriteText` in their loop. This is more efficient than `PMDFreadLine` and `PMDFwriteLine` which will needlessly strip away and then re-append a record terminator for each line read and written.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. Line not written.
PMDF__INVSTRDES	Invalid string descriptor for line : descriptor has an invalid value in its DSC\$B_CLASS field. Line not written.

PMDFwriteSubject

Write a "Subject:" header line to a message being created.

PASCAL *status = PMDF_write_subject (nq_context, subject)*

argument information

Argument	Data type	Access	Mechanism
nq_context	context pointer	read/write	reference
subject	descriptor	read	reference

C *status = PMDFwriteSubject
(nq_context, subject, subject_len)*

argument information

```
int PMDFwriteSubject(PMDF_nq **nq_context,  
                    char *subject,  
                    int subject_len)
```

ARGUMENTS

nq_context

A message enqueue context created with PMDFstartMessageEnvelope.

subject

Text to place in a "Subject:" header line; should not include the leading "Subject:". Length can not exceed 65,535 bytes.

subject_len

Length in bytes of **subject**.

DESCRIPTION

PMDFwriteSubject is a convenience routine for writing a "Subject:" header line to a message. The call

```
PMDFwriteSubject(nq_context, "Meeting at 10:30");
```

is equivalent to the call

```
PMDFwriteLine(nq_context, "Subject: Meeting at 10:30");
```

PMDFwriteSubject should be called after PMDFstartMessageHeader and prior to calling PMDFstartMessageBody. If it is called after PMDFstartMessageBody, then it's output will become part of the message body.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. "Subject:" header line not written.
PMDF__INVSTRDES	Invalid string descriptor for subject : descriptor has an invalid value in its DSC\$B_CLASS field. No "Subject:" line written.

PMDFwriteText

Write a line of text to a message being enqueued.

PASCAL *status* = **PMDF_write_text** (*nq_context*, *text*)

argument information

Argument	Data type	Access	Mechanism
<i>nq_context</i>	context pointer	read/write	reference
<i>text</i>	descriptor	read	reference

C *status* = **PMDFwriteText** (*nq_context*, *text*, *text_len*)

argument information

```
int PMDFwriteText(PMDF_nq **nq_context,  
                  char      *text,  
                  int       text_len)
```

ARGUMENTS

nq_context

A message enqueue context created with `PMDFstartMessageEnvelope`.

text

String of text to write to the message. Can not exceed a length of 65,535 bytes.

text_len

Length in bytes of ***text***.

DESCRIPTION

Text can be written to a message using `PMDFwriteText` or `PMDFwriteLine`. The only difference between these two routines is that `PMDFwriteLine` always appends a record terminator, line feed, to the end of each line it outputs. `PMDFwriteText` does not: it is left to callers of `PMDFwriteText` to include record terminators, where appropriate, in their text. This gives slightly more flexibility than `PMDFwriteLine` in that a single call can output multiple lines or multiple calls can output a single line. Note that each distinct line (record) in a message must be terminated with a line feed. If this terminator is omitted then, in the message being composed, the subsequent line will be appended directly to the end of the line lacking a terminator. While this is merely a nuisance in a message body, it can introduce serious errors into the message header.

Programs which loop reading lines from a queued message and writing them to a new message should use `PMDFreadText` and `PMDFwriteText` in their loop. This

is more efficient than PMDFreadLine and PMDFwriteLine which will needlessly strip away and then re-append a record terminator for each line read and written.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__BADCONTEXT	Illegal or corrupt context. Text not written.
PMDF__INVSTRDES	Invalid string descriptor for text : descriptor has an invalid value in its DSC\$B_CLASS field. Text not written.

2 Callable SEND

PMDF's callable send facility is a single procedure, `PMDF_send`, which can be used to send (enqueue) mail messages of local origin; that is, to originate mail from the local host. Because the callable SEND routine is not as flexible as the API routines and will take possibly undesirable, but necessary, authentication steps, the PMDF API routines should generally be used by programs which need to re-send, forward, gateway, or otherwise route mail messages.¹

Note: Callable send can be used simultaneously with the PMDF API routines.

2.1 Sending a Message

For each message to be sent with `PMDF_send`, an item list describing the message to be sent must first be built. The entries in this item list specify the message's "From:" and "To:" addresses as well as input sources for the content of the message. The basic steps in sending a message with callable send are:

1. Build the item list to pass to `PMDF_send`:
 - a. specify any special processing options (*e.g.*, `PMDF_BLANK`, `PMDF_NOIGNORE_ERRORS`, *etc.*);
 - b. specify the message's envelope "From:" address with the `PMDF_USER` item;
 - c. specify the message's "To:", "Cc:", and "Bcc:" addresses with the `PMDF_TO`, `PMDF_CC`, and `PMDF_BCC` items;
 - d. an initial message header can be specified through an input source which supplies each of the initial message header lines (`PMDF_HDR_FILE`, `PMDF_HDR_PROC`), or the content of individual message header lines can be specified with individual item codes (`PMDF_SUBJECT`, `PMDF_REPLY_TO`, `PMDF_ORGANIZATION`, *etc.*);
 - e. specify the input sources for the message body with the `PMDF_MSG_FILE` or `PMDF_MSG_PROC` items; and then
 - f. terminate the item list with `PMDF_END_LIST`.
2. Pass the item list to `PMDF_send`.
3. Check the return status from `PMDF_send`.

To enqueue additional messages, simply repeat these steps. The entire set of available item codes and their usage is given in Section 2.7.

¹ An example of such an authentication step would be the addition of a "Sender:" header line.

Callable SEND

Sending a Message

2.1.1 Envelope & Header "From:" Address

The envelope "From:" address for a message to be sent should be specified with the `PMDF_USER` item code. With this item code, only a user name can be specified; `PMDF_send` will automatically append the official local host name to the name so as to produce a valid address.

The `PMDF_ENV_FROM` item can be used to specify an envelope "From:" address which is not a local address. This is usually not necessary: applications which enqueue non-local mail should probably be using the API routines rather than callable send.

If neither `PMDF_USER` or `PMDF_ENV_FROM` are specified, then the user name associated with the current process will be used for the envelope "From:" address. When `PMDF_USER` is used, the "From:" header line address will be derived from the envelope "From:" address; when `PMDF_ENV_FROM` is used, the "From:" header line will be derived from the user name of the current process.² Only privileged users can specify with `PMDF_USER` a user name different than that of the current process's. On OpenVMS systems, `WORLD` privilege as a default privilege is required. On UNIX systems, the process must have the same (real) UID as either the `root` or `pmdf` account. On NT systems, `PMDF_send` can only be used by privileged accounts such as Administrator.

2.1.2 To:, Cc:, and Bcc: Addresses

The list of "To:", "Cc:" and "Bcc:" addresses to send a message to is built up, one address at a time, with item list entries. Each item list entry specifies the type of address ("To:", "Cc:", or "Bcc:") and a string containing the address. The type of address is denoted by the item code, `PMDF_TO`, `PMDF_CC`, or `PMDF_BCC`, associated with the item entry. `PMDF_send` will use this information to build the message's envelope "To:" address list and "To:", "Cc:", and "Bcc:" header lines.

To specify an envelope-only address which should not appear in the message header (*i.e.*, an active transport address), use `PMDF_ENV_TO`, `PMDF_ENV_CC`, or `PMDF_ENV_BCC`, as appropriate.³ Likewise, to specify a header-only address which should not appear in the envelope (*i.e.*, an inactive address), use `PMDF_HDR_TO`, `PMDF_HDR_CC`, or `PMDF_HDR_BCC`, as appropriate.

When one or more of the "To:", "Cc:", or "Bcc:" addresses is illegal, `PMDF_send` will not, by default, indicate which addresses were in error. By using the `PMDF_ADDRESS_STATUS` item code; however, this differentiation can be achieved. When this item code is used, the string containing each "To:", "Cc:", or "Bcc:" address passed in to `PMDF_send` must have a length of at least `ALFA_SIZE` bytes. On output, `PMDF_send` will overwrite each address with a status message (which includes the original address in the message). The `item_length` field associated with each address will contain the

² In either case, if a "From:" header line is supplied in an initial header, then a "Sender:" header line will be added to the message header. The initial "From:" header line will be left intact and the address specified and "Sender:" address will be derived from either the envelope "From:" address (`PMDF_USER`) or from the user name of the current process (`PMDF_ENV_FROM`).

³ While it is correct that `PMDF` currently does not distinguish between "To:", "Cc:", and "Bcc:" recipients in the envelope, distinct item codes are nonetheless provided for specifying envelope-only recipients. Use them as you see fit.

length of the returned message and an indication as to whether the address was legal or illegal. The magnitude of the value stored in the `item_length` field will give the length of the message; the sign of the value will indicate if it was legal (positive sign) or illegal (negative sign).

2.1.3 Message Headers & Content

The body of a message (*i.e.*, the message content) to be sent is built up from zero or more input files or procedures. The input files and procedures are read or invoked in the order specified in the item list passed to `PMDF_send` and the message body built up by appending the next input source to the end of the previous input source. A blank line will be inserted in the message as a separator between input sources if the `PMDF_BLANK` item is requested in the item list. The `PMDF_MSG_FILE` and `PMDF_MSG_PROC` items are used to specify the name or address of input files or procedures.

An initial message header can be supplied via either an input file or procedure. The message header will then be modified as needed when the message is enqueued. The `PMDF_HDR_FILE` and `PMDF_HDR_PROC` items are used to specify the name or address of an input file or procedure. If an initial message header is to be supplied, it must appear in the item list before any `PMDF_MSG_FILE` or `PMDF_MSG_PROC` items. A blank line must be supplied at the end of the message header or at the start of the first message body input source. This blank line will automatically be supplied when the `PMDF_BLANK` item code is specified in the item list.

The `PMDF_MODE_` and `PMDF_ENC_` items control the access mode and encodings applied to message body input sources. These items set the current access mode and encoding to be applied to all subsequent input sources which appear in the item list. The default access mode is `PMDF_MODE_UNKNOWN` which uses a text mode access and the default encoding is `PMDF_ENC_UNKNOWN` which results in no encoding of the data. The block access mode will not be applied to input procedures; the access mode and encodings do not apply to input sources for an initial message header which is always accessed using the default access mode and never encoded.

Input procedures use the calling format:

status = proc (**bufadr**, **buflen**)

where

bufadr is the address of a buffer to receive the next line of input. **bufadr** is passed by value.

buflen is an integer which, on input, specifies the maximum size in bytes of the buffer pointed at by **bufadr** and, on output, receives the length of the data read into that buffer. **buflen** is passed by reference.

The return value **status** is an integer which should be set to zero (0) when there are no more lines to return and one (1) at all other times including when the last line itself is returned.

Callable SEND

Sending a Message

2.2 Writing Output from a Channel Program

The `stdin`, `stdout`, and `stderr` I/O destinations (`SYSS$INPUT`, `SYSS$OUTPUT`, and `SYSS$ERROR`) are all controlled by PMDF and will vary depending upon the context under which a channel program has been invoked. As such, programs which will operate as PMDF channels should use the `PMDFlog` routine described in Chapter 1 to write information to their log file. Such programs should never write output directly to `stdout` or `stderr` or other generic I/O destinations (*e.g.*, Pascal's "output" or FORTRAN's default output logical unit). There's no telling where such output might go: it might go to the job controller's log file, it might even go down a network pipe to a remote client or server.

Note that the channel log file is a different file than the PMDF log file; the `PMDF_log` and `PMDF_close_log_file` are unrelated routines.

2.3 Required Privileges

Like the PMDF API routines, privileges are required in order to use callable SEND. Enqueuing messages requires privileges sufficient to create, open, read from, and write to the queue cache database as well as to create subdirectories and files in the PMDF message queue directories. There are any number of ways of accomplishing this under OpenVMS; the typical being to have the program run under the `SYSTEM` account. On UNIX, this is accomplished by having your executable program owned and run by the `pmdf` account or, alternatively, owned by `pmdf` and have the `setuid` attribute set. On NT systems, `PMDF_send` can only be used by privileged accounts such as Administrator.

In order to submit mail under a user name which differs from that of the calling process, privileges are required. On OpenVMS, `WORLD` default privilege is needed. On UNIX, the process must have the same (real) UID as either the `root` or `pmdf` account. On NT, the process must be a privileged account such as Administrator.

In addition, under OpenVMS the account running your program must have `SYSPRV` and `CMKRNL` privileges. These privileges are required so that PMDF can submit any processing jobs required to handle an enqueued message. Note that PMDF itself does not use these privileges: they are required by the `$SNDJBC` system service call used to dispatch processing jobs.

In some applications, it is important to keep strict control over when privileges are enabled and disabled. To this end, the `PMDF_PRIV_ENABLE_PROC` and `PMDF_PRIV_DISABLE_PROC` item codes can be used to specify the addresses of two procedures to call immediately prior to and immediately after enqueueing a message. This allows the required privileges to be enabled only when they are needed — when the message is enqueued — and to remain disabled at all other times. Callable SEND does not use a condition handler, so if a fatal error occurs while enqueueing a message, it is up to the calling program to trap the error and, if necessary, disable any privileges which should be disabled. These procedures, if specified, should accept no arguments and return no return value (*i.e.*, function result).

The privileges to be enabled must either be granted to the program using callable SEND (*e.g.*, the program can be installed with privileges) or the process running the program must have the requisite privileges. Callable SEND and PMDF in no way provide these privileges.

2.4 Compiling and Linking Programs

Programs which use callable SEND are linked using the same steps as the API routines. Refer to Section 1.11 for details.

2.5 Examples of Using Callable SEND

Several example programs, written in Pascal and C, are provided in this section:

- Examples 2-1, 2-2, and 2-3 illustrate sending a simple message;
- Examples 2-4 — 2-7 illustrate specifying an initial message header;
- Examples 2-8, 2-9, and 2-10 illustrate sending a message to multiple recipients (To:, and Cc:) as well as to FAX addresses (To: and Bcc:) and examining returned status messages for each address; and
- Examples 2-11 and 2-12 illustrate the use of an input procedure to generate the body of the message to be sent.

The example routines shown in this section can be found, on OpenVMS systems, in the directory, `PMDF_ROOT:[DOC.EXAMPLES]`. On UNIX and NT systems, the examples can be found in the `/pmdf/doc/examples` directory.

Note: The example Pascal programs are intended for use on OpenVMS. To use them on UNIX or NT, changes to the examples will be required.

2.5.1 Sending a Simple Message

The programs shown in Examples 2-1 and 2-2 demonstrate how to send a simple message to the SYSTEM account. The caller's login command procedure is used as the input source for the body of the message to be sent. The From: address associated with the message is that of the process running the program. The output of these programs is given in Example 2-3. The callouts shown in the first two examples produce the corresponding output shown in the third example.

Example 2-1 Sending a Simple Message (Pascal)

Example 2-1 Cont'd on next page

Callable SEND

Examples of Using Callable SEND

Example 2-1 (Cont.) Sending a Simple Message (Pascal)

```
(* send_example1.pas -- Send a simple message *)
[inherit ('pmdf_exe:apidef')] program send_example1;
var
  item_index : integer := 0;
  item_list  : array [1..4] of PMDF_item_list;
  msgfile    : varying [20] of char := 'SYS$LOGIN:LOGIN.COM';
  subject    : varying [20] of char := 'Your login.com file';
  to_adr     : varying [20] of char := 'SYSTEM';

function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;

(* Push an string oriented entry onto the item list *)
procedure push_str (code : integer; var str : varying [len] of char);

begin (* push_str *)
  item_index := succ (item_index);
  with item_list[item_index] do begin
    item_code := code;
    item_address := (iaddress (str.body))::$stringptr;
    item_length := str.length;
  end; (* with *)
end; (* push_str *)

begin (* send_example1 *)
  push_str (PMDF_TO, to_adr); ①
  push_str (PMDF_SUBJECT, subject); ②
  push_str (PMDF_MSG_FILE, msgfile); ③
  item_list[item_index+1].item_code := 0;
  SYS$EXIT (PMDF_send ((iaddress (item_list))::PMDF_item_list_ptr));
end. (* send_example1 *)
```

Example 2-2 Sending a Simple Message (C)

```
/* send_example2.c -- Send a simple message */
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif

/* Push an entry onto the item list */
#define ITEM(item,adr,len) item_list[item_index].item_code = (item); \
                          item_list[item_index].item_address = (char *) (adr); \
                          item_list[item_index].item_length = (len); \
                          item_index++
```

Example 2-2 Cont'd on next page

Example 2–2 (Cont.) Sending a Simple Message (C)

```
main ()
{
    PMDF_item_list item_list[4];
    int item_index = 0;
    char *subject = "Your login procedure";
#ifdef __VMS
    char *toadr = "system";
    char *msgfile = "sys$login:login.com";
#else
    char *toadr = "root";
    char *msgfile = "~/login";
#endif

    ITEM (PMDF_TO, toadr, strlen (toadr)); ❶
    ITEM (PMDF_SUBJECT, subject, strlen (subject)); ❷
    ITEM (PMDF_MSG_FILE, msgfile, strlen (msgfile)); ❸
    ITEM (PMDF_END_LIST, 0, 0);
    exit (PMDF_send (&item_list));
}
```

Example 2–3 Output of Examples 2–1 and 2–2

```
Date: 04 Oct 2012 22:24:07 -0700 (PDT)
From: dominic@yourstruely.com
Subject: Your login procedure ❷
To: system@yourstruely.com ❶
Message-id: <01GPKF10JIB89LV1WX@yourstruely.com>
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT

$ reply/enable=(tapes) ❸
$ set terminal/insert
$ define/job dbg$decw$display " "
$ @mathlib_tools:login.com
```

2.5.2 Specifying an Initial Message Header

The programs shown in Examples 2–4 and 2–5 illustrate the use of the PMDF_HDRMSG_FILE and PMDF_HDR_ADDRS item codes to enqueue a message which has already been composed — headers and all — and stored in a file. Example 2–6 shows input file. The resulting message is shown in Example 2–7.

When the entire message, header and body, is contained in a single file, use the PMDF_HDRMSG_FILE item code in place of the PMDF_HDR_FILE and PMDF_MSG_FILE item codes.

Callable SEND

Examples of Using Callable SEND

Example 2-4 Specifying an Initial Message Header (Pascal)

```
(* send_example3.pas -- Send a message with initial header *)
[inherit ('pmdf_exe:apidef')] program send_example3;

var
  item_index : integer := 0;
  item_list  : array [1..3] of PMDF_item_list;
  msgfile    : varying [40] of char := 'PMDF_ROOT:[DOC.EXAMPLES]EXAMPLE.TXT';

function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;

(* Push an option oriented entry onto the item list *)
procedure push_opt (code : integer);

begin (* push_opt *)
  item_index := succ (item_index);
  with item_list[item_index] do begin
    item_code   := code;
    item_address := nil;
    item_length := 0;
  end; (* with *)
end; (* push_opt *)

(* Push a string oriented entry onto the item list *)
procedure push_str (code : integer; var str : varying [len] of char);

begin (* push_str *)
  item_index := succ (item_index);
  with item_list[item_index] do begin
    item_code   := code;
    item_address := (iaddress (str.body))::$stringptr;
    item_length := str.length;
  end; (* with *)
end; (* push_str *)

begin (* send_example3 *)
  push_opt (PMDF_HDR_ADDRS);
  push_str (PMDF_HDRMSG_FILE, msgfile);
  push_opt (PMDF_END_LIST);
  SYS$EXIT (PMDF_send ((iaddress (item_list))::PMDF_item_list_ptr));
end. (* send_example3 *)
```

Example 2-5 Specifying an Initial Message Header (C)

```
/* send_example4.c -- Send a message with initial header */
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif
```

Example 2-5 Cont'd on next page

Callable SEND

Examples of Using Callable SEND

Example 2-5 (Cont.) Specifying an Initial Message Header (C)

```
/* Push an entry onto the item list */
#define ITEM(item,adr,len) item_list[item_index].item_code    = (item);      \
                          item_list[item_index].item_address = (char *) (adr); \
                          item_list[item_index].item_length  = (len);      \
                          item_index++

main ()
{
    PMDF_item_list item_list[3];
    int item_index = 0;
#ifdef __VMS
    char *msgfile = "PMDF_ROOT:[DOC.EXAMPLES]EXAMPLE.TXT";
#else
    char *msgfile = "/pmdf/doc/examples/example.txt";
#endif

    ITEM (PMDF_HDR_ADDRS, 0, 0);
    ITEM (PMDF_HDRMSG_FILE, msgfile, strlen (msgfile));
    ITEM (PMDF_END_LIST, 0, 0);
    exit (PMDF_send (&item_list));
}
```

Example 2-6 Input File Used in Examples 2-4 and 2-5

```
Subject: PMDF callable SEND example
To: system@sigurd.yourstruely.com
Message-id: <01GPKFNPUQF89LV1WX@sigurd.yourstruely.com>
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT
Comments: Ignore this message -- it's just a test

This is a test of the emergency broadcasting system!

123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
000000000111111111122222222223333333333344444444445555555555666666666677777777778
```

Callable SEND

Examples of Using Callable SEND

Example 2-7 Output of Examples 2-4 and 2-5

```
Date: 04 Oct 2012 22:42:25 -0700 (PDT)
From: system@sigurd.yourstruely.com
Subject: PMDF callable SEND example
To: system@sigurd.yourstruely.com
Message-id: <01GPKFNPUQF89LV1WX@sigurd.yourstruely.com>
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT
Comments: Ignore this message -- it's just a test

This is a test of the emergency broadcasting system!

123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
00000000011111111112222222222333333333333333344444444445555555555666666666677777777778
```

2.5.3 Multiple Recipients, FAX Addresses, and Per Address Status Messages

The programs given in Examples 2-8 and 2-9 demonstrate three concepts:

1. sending a message to multiple recipients;
2. constructing FAX addresses; and
3. obtaining the status (legal, illegal) of each To:, Cc:, and Bcc: address.

The message is sent to three To: addresses, one of which is a FAX address, a Cc: address, and a FAX Bcc: address. After PMDF_send is called, any status message associated with each address is displayed. This information is only displayed if PMD_send either returned a successful status code or a PMDF__HOST error. In any other case, the status messages can not have been set. The terminal output produced by running the programs is shown in Example 2-10.

The following items of note are identified with callouts in each of the two programs:

- ❶ The status of the regular (*i.e.*, non FAX) addresses will be output in the same strings used to input the addresses.
- ❷ The status of the FAX addresses will be stored in a string specified with the PMDF_FAX_TO (❸) or PMDF_FAX_BCC (❹) item codes.
- ❺ Instruct PMDF_send to return a status message for each To:, Cc:, and Bcc: address.
- ❻ Specify some To: and Cc: addresses.
- ❼ Begin a FAX To: address. Any status message for this address will be returned in the fax_adr_1 string.
- ❽ Begin a FAX Cc: address. Any status message for this address will be returned in the fax_adr_2 string.
- ❾ Attempt to send the message.
- ❿ Display any returned status messages. (See Example 2-10.)

Example 2-8 Multiple Addresses (Pascal)

```
(* send_example5.pas -- Send a message to multiple recipients,
    including FAX recipients *)

[inherit ('pmdf_exe:apidef')] program send_example5 (output);
  type string = varying [ALFA_SIZE] of char;
  var
    item_index : integer := 0; i, stat : integer;
    item_list  : array [1..19] of PMDF_item_list;
    msgfile    : string := 'sys$login:login.com';
    subject    : string := 'PMDF callable SEND example: sending a FAX';
    to_adr_1   : [static] string := 'system'; ❶
    to_adr_2   : [static] string := 'bob@example.com'; ❶
    cc_adr_1   : [static] string := 'sue@example.com'; ❶

    (* First FAX address *)
    fn_1       : string := '1-714-555-5319';           (* REQUIRED *)
    domain_1   : string := 'text-fax.example.com';     (* REQUIRED *)
    at_1       : string := 'Mrochek Freed';
    o_1        : string := 'Example Software, LLC';
    ou1_1      : string := '9 Main Street';
    ou2_1      : string := 'Springfield, USA';
    tn_1       : string := '(508) 555-1111';
    fax_adr_1  : [static] string; ❷

    (* Second FAX address *)
    fn_2       : string := '1-800-555-1212';           (* REQUIRED *)
    domain_2   : string := 'text-fax.example.com';     (* REQUIRED *)
    at_2       : string := '800 Directory Assistance';
    fax_adr_2  : [static] string; ❷

function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;

(* Push an option oriented entry onto the item list *)
procedure push_opt (code : integer);
  begin (* push_opt *)
    item_index := succ (item_index);
    with item_list[item_index] do begin
      item_code := code;
      item_address := nil;
      item_length := 0;
    end; (* with *)
  end; (* push_opt *)

(* Push an string oriented entry onto the item list *)
procedure push_str (code : integer; var str : varying [len] of char);
  begin (* push_str *)
    item_index := succ (item_index);
    with item_list[item_index] do begin
      item_code := code;
      item_address := (iaddress (str.body))::$stringptr;
    end;
  end;
end;
```

Example 2-8 Cont'd on next page

Callable SEND

Examples of Using Callable SEND

Example 2-8 (Cont.) Multiple Addresses (Pascal)

```
    item_length := str.length;
    end; (* with *)
    end; (* push_str *)

begin (* send_example5 *)

    (* Specify the Subject: header line and message input source *)
    push_str (PMDF_SUBJECT,      subject);
    push_str (PMDF_MSG_FILE,     msgfile);

    (* Return per address status/error messages *)
    push_opt (PMDF_ADDRESS_STATUS); ③

    (* Specify regular To: and Cc: addresses *)
    push_str (PMDF_TO,          to_adr_1); ④
    push_str (PMDF_TO,          to_adr_2); ④
    push_str (PMDF_CC,          cc_adr_1); ④

    (* Specify the first FAX address *)
    push_str (PMDF_FAX_TO,      fax_adr_1); ⑤
    push_str (PMDF_FAX_DOMAIN,  domain_1);
    push_str (PMDF_FAX_FN,      fn_1);
    push_str (PMDF_FAX_AT,      at_1);
    push_str (PMDF_FAX_O,       o_1);
    push_str (PMDF_FAX_OU,      ou1_1);
    push_str (PMDF_FAX_OU,      ou2_1);
    push_str (PMDF_FAX_TN,      tn_1);

    (* Specify the second FAX address *)
    push_str (PMDF_FAX_BCC,     fax_adr_2); ⑥
    push_str (PMDF_FAX_DOMAIN,  domain_2);
    push_str (PMDF_FAX_FN,      fn_2);
    push_str (PMDF_FAX_AT,      at_2);

    (* Now terminate the item list *)
    push_opt (PMDF_END_LIST);

    (* And send the message *)
    stat := PMDF_send ((iaddress (item_list))::PMDF_item_list_ptr); ⑦

    (* Display the address status messages provided that no error
       other than PMDF__HOST has occurred. *)
    if odd (stat) or (stat = PMDF__HOST) then for i := 1 to item_index do ⑧
    with item_list[i] do case item_code of
        PMDF_TO,      PMDF_CC,      PMDF_BCC,
        PMDF_FAX_TO, PMDF_FAX_CC, PMDF_FAX_BCC,
        PMDF_PRT_TO, PMDF_PRT_CC, PMDF_PRT_BCC :
            writeln (substr (item_address^, 1, abs (item_length)));
        otherwise begin end;
    end; (* case, with, for, if *)

    (* Now exit *)
    SYS$EXIT (stat);

end. (* send_example5 *)
```

Example 2-9 Multiple Addresses (C)

```
/* send_example6.c -- Send a message to multiple recipients,
   including FAX recipients */

#include <stdio.h>
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif

/* Push an entry onto the item list */
#define ITEM(item,adr,len) item_list[item_index].item_code    = (item);      \
                           item_list[item_index].item_address = (char *) (adr); \
                           item_list[item_index].item_length  = (len);      \
                           item_index++

main ()
{
    int item_index = 0, stat;
    PMDF_item_list item_list[19];
    char *subject  = "PMDF callable SEND example: sending a FAX";
    char to_adr_2[ALFA_SIZE+1] = "bob@example.com"; ❶
    char cc_adr_1[ALFA_SIZE+1] = "sue@example.com"; ❶
#ifdef __VMS
    char *msgfile = "sys$login:login.com";
    char to_adr_1[ALFA_SIZE+1] = "system";
#else
    char *msgfile = "~/login";
    char to_adr_1[ALFA_SIZE+1] = "root";
#endif

    /* First FAX address */
    char *fn_1      = "1-714-555-5319";              /* REQUIRED */
    char *domain_1 = "text-fax.example.com";         /* REQUIRED */
    char *at_1      = "Mrochek Freed";
    char *o_1       = "Example Software, LLC";
    char *ou1_1     = "9 Main Street";
    char *ou2_1     = "Springfield, USA";
    char *tn_1      = "(508) 555-1111";
    char fax_adr_1[ALFA_SIZE+1]; ❷

    /* Second FAX address */
    char *fn_2      = "1-800-555-1212";              /* REQUIRED */
    char *domain_2  = "text-fax.example.com";         /* REQUIRED */
    char *at_2      = "800 Directory Assistance";
    char fax_adr_2[ALFA_SIZE+1]; ❷

    /* Specify the Subject: header line and message input source */
    ITEM (PMDF_SUBJECT,    subject,    strlen (subject));
    ITEM (PMDF_MSG_FILE,   msgfile,    strlen (msgfile));

    /* Return per address status/error messages */
    ITEM (PMDF_ADDRESS_STATUS, 0,      0); ❸
}
```

Example 2-9 Cont'd on next page

Callable SEND

Examples of Using Callable SEND

Example 2-9 (Cont.) Multiple Addresses (C)

```
/* Specify regular To: and Cc: addresses */
ITEM (PMDF_TO,                to_adr_1,  strlen (to_adr_1)); ④
ITEM (PMDF_TO,                to_adr_2,  strlen (to_adr_2)); ④
ITEM (PMDF_CC,                cc_adr_1,  strlen (cc_adr_1)); ④

/* Specify the first FAX address */
ITEM (PMDF_FAX_TO,           fax_adr_1,  0); ⑤
ITEM (PMDF_FAX_DOMAIN,      domain_1,  strlen (domain_1));
ITEM (PMDF_FAX_FN,          fn_1,      strlen (fn_1));
ITEM (PMDF_FAX_AT,          at_1,      strlen (at_1));
ITEM (PMDF_FAX_O,           o_1,       strlen (o_1));
ITEM (PMDF_FAX_OU,          ou1_1,     strlen (ou1_1));
ITEM (PMDF_FAX_OU,          ou2_1,     strlen (ou2_1));
ITEM (PMDF_FAX_TN,          tn_1,      strlen (tn_1));

/* Specify the second FAX address */
ITEM (PMDF_FAX_BCC,         fax_adr_2,  0); ⑥
ITEM (PMDF_FAX_DOMAIN,      domain_2,  strlen (domain_2));
ITEM (PMDF_FAX_FN,          fn_2,      strlen (fn_2));
ITEM (PMDF_FAX_AT,          at_2,      strlen (at_2));

/* Now terminate the item list */
ITEM (PMDF_END_LIST,        0,          0);

/* And send the message */
stat = PMDF_send (&item_list); ⑦

/* Display the address status messages provided that no error
   other than PMDF__HOST has occurred. */
if ((1 & stat) || stat == PMDF__HOST) { ⑧
    int i, j;
    for (i = 0; i < item_index; i++) {
        switch (item_list[i].item_code) {
            case PMDF_TO      : case PMDF_CC      : case PMDF_BCC      :
            case PMDF_FAX_TO : case PMDF_FAX_CC : case PMDF_FAX_BCC :
            case PMDF_PRT_TO  : case PMDF_PRT_CC  : case PMDF_PRT_BCC :
                j = abs (item_list[i].item_length);
                item_list[i].item_address[j] = '\0';
                printf ("%s\n", item_list[i].item_address);
                break;
            default : break;
        }
    }
}
exit (stat);
}
```

Example 2-10 Address Status Messages Produced by Examples 2-8 and 2-9

```
address okay: system
address okay: bob@example.com
```

Example 2-10 Cont'd on next page

Example 2-10 (Cont.) Address Status Messages Produced by Examples 2-8 and 2-9

```
address okay: sue@example.com
address okay: "/FN=1-714-555-5319/AT=John Jones/O=Example Software, LLC
./OU=9 Main Street/OU=Springfield, USA/TN=(508) 555-1111/"
@text-fax.example.com
address okay: "/FN=1-800-555-1212/AT=800 Directory Assistance/"
@text-fax.example.com
```

2.5.4 Using an Input Procedure

The programs shown in Examples 2-11 and 2-12 use an input procedure as the source for the body of a message to be sent. In the Pascal program example, the procedure `msg_proc` will continue to read input until a blank line is entered at which point the message will be sent. In the C program example, the input procedure `msg_proc` will read input until the run-time library routine `fgets()` signals an EOF (e.g., a control-Z has been input). In both programs, the address of the procedure `msg_proc` is passed to `PMDF_send` via a `PMDF_MSG_PROC` item code and `PMDF_send` itself repeatedly calls the procedure until a value of 0 is returned by the procedure.

Example 2-11 Using an Input Procedure (Pascal)

```
(* send_example7.pas -- Demonstrate the use of PMDF_MSG_PROC *)
[inherit ('pmdf_exe:apidef')] program send_example7 (input, output);

type
  string      = varying [ALFA_SIZE] of char;
  string_ptr  = ^string;

var
  item_index : integer := 0;
  item_list  : array [1..4] of PMDF_item_list;
  subject    : string := 'PMDF callable SEND example';
  to_adr     : string := 'system';

function SYS$EXIT (%immed status : integer := %immed 1) : integer; extern;

(* Push an entry onto the item list *)
procedure push (code : integer; adr : unsigned; len : integer);

begin (* push *)
  item_index := succ (item_index);
  with item_list[item_index] do begin
    item_code      := code;
    item_address   := adr::$string_ptr;
    item_length    := len;
  end; (* with *)
end; (* push *)
```

Example 2-11 Cont'd on next page

Callable SEND

Examples of Using Callable SEND

Example 2-11 (Cont.) Using an Input Procedure (Pascal)

```
function msg_proc (var str_i : integer; var str_len : integer) : integer;
  type
    chars    = packed array [1..BIGALFA_SIZE] of char;
    char_ptr = ^chars;

  var
    buffer : string;
    i      : integer;
    str    : char_ptr;

  begin (* msg_proc *)
    write ('input: ');
    readln (buffer);
    if buffer.length = 0 then begin
      str_len := 0;
      msg_proc := 0;
    end else begin
      str := (iaddress (str_i))::char_ptr;
      str_len := min (buffer.length, str_len);
      for i := 1 to str_len do str^[i] := buffer[i];
      msg_proc := 1;
    end; (* if *)
  end; (* msg_proc *)

begin (* send_example7 *)
  push (PMDF_SUBJECT, iaddress (subject.body), subject.length);
  push (PMDF_TO, iaddress (to_adr.body), to_adr.length);
  push (PMDF_MSG_PROC, iaddress (msg_proc), 4);
  push (PMDF_END_LIST, 0, 0);
  SYS$EXIT (PMDF_send ((iaddress (item_list))::PMDF_item_list_ptr));
end. (* send_example7 *)
```

Example 2-12 Using an Input Procedure (C)

```
/* send_example8.c -- Demonstrate the use of PMDF_MSG_PROC */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __VMS
#include "pmdf_com:apidef.h"
#else
#include "/pmdf/include/apidef.h"
#endif

/* Push an entry onto the item list */
#define ITEM(item,adr,len) item_list[item_index].item_code    = (item);      \
                          item_list[item_index].item_address = (char *) (adr); \
                          item_list[item_index].item_length  = (len);      \
                          item_index++
```

Example 2-12 Cont'd on next page

Example 2-12 (Cont.) Using an Input Procedure (C)

```
int msg_proc (char *str, int *str_len)
{
    printf ("input: ");
    if (fgets (str, *str_len, stdin)) {
        *str_len = strlen (str);
        if (str[*str_len-1] == '\n') *str_len -= 1;
        return (1);
    }
    else {
        *str_len = 0;
        return (0);
    }
}

main ()
{
    int istat, item_index = 0;
    PMDF_item_list item_list[4];
    char *subject = "PMDF callable SEND example";
#ifdef _VMS
    char *to_adr = "system";
#else
    char *to_adr = "root";
#endif

    if (!(1 & (istat = PMDFinitialize (0)))) exit (istat);
    ITEM (PMDF_SUBJECT, subject, strlen (subject));
    ITEM (PMDF_TO, to_adr, strlen (to_adr));
    ITEM (PMDF_MSG_PROC, msg_proc, 4);
    ITEM (PMDF_END_LIST, 0, 0);
    exit (PMDF_send (item_list));
}
```

2.6 Summary of PMDF_send Item Codes

A summary of the PMDF_send item codes is given in Table 2-1.

Callable SEND

Summary of PMDF_send Item Codes

Table 2-1 PMDF_send Item Code Summary

Addressing item codes	Description
PMDF_BCC	Specify a Bcc: address which will appear in the header and envelope
PMDF_CC	Specify a cc: address which will appear in the header and envelope
PMDF_ENV_BCC	Specify an envelope-only Bcc: address
PMDF_ENV_CC	Specify an envelope-only Cc: address
PMDF_ENV_FROM	Specify the envelope From: address
PMDF_ENV_TO	Specify an envelope-only To: address
PMDF_HDR_ADDRS	Obtain recipient addresses from the RFC 822 header
PMDF_HDR_BCC	Specify a header-only Bcc: address
PMDF_HDR_CC	Specify a header-only Cc: address
PMDF_HDR_NOADDRS	Do not obtain recipient addresses from the RFC 822 header
PMDF_HDR_TO	Specify a header-only To: address
PMDF_TO	Specify a To: address which will appear in the header and envelope
PMDF_SUBADDRESS	Specify a subaddress
PMDF_USER	Specify the user name to use for the envelope From: and header line From: addresses
<hr/>	
FAX addressing item codes	Description
PMDF_FAX_AFTER	FAX address AFTER attribute-value pair
PMDF_FAX_AT	FAX address AT attribute-value pair
PMDF_FAX_AUTH	FAX address AUTH attribute-value pair
PMDF_FAX_BCC	Begin the specification of a FAX Bcc: address
PMDF_FAX_CC	Begin the specification of a FAX Cc: address
PMDF_FAX_DOMAIN	Domain name to associate with a FAX address
PMDF_FAX_FN	FAX address FN attribute-value pair
PMDF_FAX_FSI	FAX address FSI attribute-value pair
PMDF_FAX_O	FAX address O attribute-value pair
PMDF_FAX_OU	FAX address OU attribute-value pair
PMDF_FAX_SETUP	FAX address SETUP attribute-value pair
PMDF_FAX_SFN	FAX address SFN attribute-value pair
PMDF_FAX_STN	FAX address STN attribute-value pair
PMDF_FAX_TO	Begin the specification of a FAX To: address
PMDF_FAX_TN	FAX address TN attribute-value pair
PMDF_FAX_TTI	FAX address TTI attribute-value pair

Callable SEND Summary of PMDF_send Item Codes

Table 2–1 (Cont.) PMDF_send Item Code Summary

Printer addressing item codes	Description
Printer addressing	
item codes	
PMDF_PRT_AT	Printer address AT attribute-value pair
PMDF_PRT_BCC	Begin the specification of a printer Bcc: address
PMDF_PRT_CC	Begin the specification of a printer Cc: address
PMDF_PRT_DOMAIN	Specify the domain name to associate with a printer address
PMDF_PRT_MS	Printer address MS attribute-value pair
PMDF_PRT_O	Printer address O attribute-value pair
PMDF_PRT_OU	Printer address OU attribute-value pair
PMDF_PRT_P1	Printer address P1 attribute-value pair
PMDF_PRT_P2	Printer address P2 attribute-value pair
PMDF_PRT_P3	Printer address P3 attribute-value pair
PMDF_PRT_P4	Printer address P4 attribute-value pair
PMDF_PRT_P5	Printer address P5 attribute-value pair
PMDF_PRT_P6	Printer address P6 attribute-value pair
PMDF_PRT_P7	Printer address P7 attribute-value pair
PMDF_PRT_P8	Printer address P8 attribute-value pair
PMDF_PRT_TO	Begin the specification of a printer To: address
PMDF_PRT_TN	Printer address TN attribute-value pair
Header processing	
item codes	
PMDF_COMMENTS	Specify the body of a Comments: header line
PMDF_CONTENT_TYPE	Specify the body of a Content-type: header line
PMDF_DELIVERY_RECEIPT_TO	Specify the body of a Delivery-receipt-to: header line
PMDF_ERRORS_TO	Specify the body of an Errors-to: header line
PMDF_EXTRA_HEADER	Specify an additional header line
PMDF_FROM	Specify the body of a From: header line
PMDF_FRUIT_OF_THE_DAY	Specify the body of a Fruit-of-the-day: header line
PMDF_HDR_ADDRS	Obtain recipient addresses from the RFC 822 header
PMDF_HDR_FILE	Specify an initial message header input source file
PMDF_HDR_NOADDRS	Do not obtain recipient addresses from the RFC 822 header
PMDF_HDR_NORESENT	Do not resort to Resent- header lines
PMDF_HDR_PROC	Specify an initial message header input source procedure
PMDF_HDR_RESENT	Add addresses to the header using Resent- header lines if necessary
PMDF_HDRMSG_FILE	Specify a file containing initial RFC 822 header information and a message body part
PMDF_HDRMSG_PROC	Specify an initial RFC 822 header and message body part input source procedure
PMDF_IMPORTANCE	Specify the body of an Importance: header line
PMDF_KEYWORDS	Specify the body of a Keywords: header line
PMDF_ORGANIZATION	Specify the body of an Organization: header line
PMDF_PRIORITY	Specify the body of a Priority: header line
PMDF_READ_RECEIPT_TO	Specify the body of a Read-receipt-to: header line

Callable SEND

Summary of PMDF_send Item Codes

Table 2-1 (Cont.) PMDF_send Item Code Summary

Header processing	
item codes	Description
PMDF_REFERENCES	Specify the body of a References: header line
PMDF_REPLY_TO	Specify the body of a Reply-to: header line
PMDF_RESENT_FROM	Specify the body of a Resent-from: header line
PMDF_RESENT_REPLY_TO	Specify the body of a Resent-reply-to: header line
PMDF_SENSITIVITY	Specify the body of a Sensitivity: header line
PMDF_SUBJECT	Specify the body of a Subject: header line
PMDF_WARNINGS_TO	Specify the body of a Warnings-to: header line
PMDF_X_ORGANIZATION	Specify the body of a X-Organization: header line
PMDF_X_PS_QUALIFIERS	Specify the body of a X-PS-Qualifiers: header line
Message body processing	
item codes	Description
PMDF_CONTENT_FILENAME	Include the file name as a parameter in the MIME Content-type: header line
PMDF_ENC_BASE64	MIME BASE64 encode the message body part
PMDF_ENC_BASE85	Adobe ASCII85 (BASE85) encode the message body part
PMDF_ENC_BINHEX	BINHEX encode the message body part
PMDF_ENC_BTOA	BTOA encode the message body part
PMDF_ENC_COMPRESSED_BASE64	GNU zip compress the message body part and then MIME BASE64 encode it
PMDF_ENC_COMPRESSED_BINARY	GNU zip compress the message body part
PMDF_ENC_COMPRESSED_UUENCODE	GNU zip compress the message body part and then UUENCODE it
PMDF_ENC_HEXADECIMAL	Hexadecimal encode the message body part
PMDF_ENC_NONE	Do not encode the message body part
PMDF_ENC_PATHWORKS	Encode the message body part using a DEC Pathworks compatible encoding
PMDF_ENC_QUOTED_PRINTABLE	MIME quoted printable encode the message body part
PMDF_ENC_UNKNOWN	Do not encode the message body part (default)
PMDF_ENC_UUENCODE	UUENCODE the message body part
PMDF_HDR_FILE	Specify a file containing initial RFC 822 header information
PMDF_HDR_PROC	Specify an initial RFC 822 header input source procedure
PMDF_HDRMSG_FILE	Specify a file containing initial RFC 822 header information and a message body part
PMDF_HDRMSG_PROC	Specify an initial RFC 822 header and message body part input source procedure
PMDF_MAX_BLOCKS	Specify the maximum number of blocks per message
PMDF_MAX_BYTES	Specify the maximum number of bytes per message
PMDF_MAX_LINES	Specify the maximum number of message lines per message
PMDF_MAX_TO	Specify the maximum number of envelope To: addresses per message copy
PMDF_MODE_BLOCK	Access input files using block mode I/O
PMDF_MODE_RECORD	Access input files using record mode I/O
PMDF_MODE_RECORD_CRATTRIBUTE	Access input files using record mode I/O & preserve <CR> record terminators

Callable SEND

Summary of PMDF_send Item Codes

Table 2-1 (Cont.) PMDF_send Item Code Summary

Message body processing	
item codes	Description
PMDF_MODE_RECORD_CRLFATTRIBUTE	Access input files using record mode I/O & preserve <CR><LF> record terminators
PMDF_MODE_RECORD_LFATTRIBUTE	Access input files using record mode I/O & preserve <LF> record terminators
PMDF_MODE_TEXT	Access input files using text mode I/O
PMDF_MODE_UNKNOWN	Access input files using text mode I/O
PMDF_MSG_FILE	Specify a message body input source file
PMDF_MSG_PROC	Specify a message body input source procedure
PMDF_NOCONTENT_FILENAME	Do not include the file name as a parameter in the MIME Content-type: header line
Miscellaneous	
item codes	Description
PMDF_BLANK	Insert a blank line between the input from each input source
PMDF_CHAIN	Pointer to another item list to process
PMDF_CHANNEL	Specify the channel to act as when enqueueing mail
PMDF_END_LIST	Terminate an item list
PMDF_IGNORE_ERRORS	Send the message as long as at least one envelope To: address and at least one input source was okay
PMDF_IS_CHANNEL	Ignore user-to-channel access checks
PMDF_IS_NOT_CHANNEL	Do not ignore user-to-channel access checks
PMDF_NOBLANK	Do not insert a blank line between each input source
PMDF_NOIGNORE_ERRORS	Send the message only if all envelope To: addresses and all input sources are okay
PMDF_PRIV_DISABLE_PROC	Privilege disable procedure to invoke after sending
PMDF_PRIV_ENABLE_PROC	Privilege enable procedure to invoke prior to sending

2.7 PMDF_send Routine Description

In the following description, the string lengths CHANLENGTH and ALFA_SIZE are mentioned. These values are defined in the API include files and listed in Table 1-2.

PMDF_send

Send a message.

FORMAT *status* = **PMDF_send** (*item_list*)

argument information

Argument	Data type	Access	Mechanism
<i>item_list</i>	item list	read	reference

ARGUMENTS

item_list

Item list specifying actions to be taken by **PMDF_send**. The **item_list** argument is the address of a list of item descriptors, each of which specifies an action and provides the information needed to perform that action. The list of item descriptors is terminated with an entry with the **PMDF_END_LIST** item code.⁴ Each item descriptor has the following C-style structure declaration:

```
struct {
    int    item_code;
    void  *item_address;
    int    reserved;
    int    item_length;
} PMDF_item_list;
```

PMDF_send Item Descriptor Fields

item_code

A longword (4 bytes) containing a user-supplied symbolic code specifying an action to be taken by **PMDF_send**. The include files described in Section 1.11 defines these codes. A description of each item code follows this list of item descriptor fields.

item_address

A longword (4 bytes) containing the user-supplied address of a character string to be used in conjunction with the action specified by the *item_code* field. Not all actions require that an *item_address* be specified.

item_length

A longword (4 bytes) containing the user-supplied length of the character string pointed at by *item_address*. Not all actions require that an *item_address* be specified.

⁴ A single longword value of zero can instead be used.

PMDF_send Item Codes

PMDF_ADDRESS_STATUS

Return status messages for each To:, Cc:, and Bcc: address specified with the PMDF_*TO, PMDF_*CC, and PMDF_*BCC item codes. When this item code is specified, all address strings specified with PMDF_*TO, PMDF_*CC, and PMDF_*BCC must have a maximum length of at least ALFA_SIZE bytes. On input to PMDF_send the string contains an address whose length is given by the associated item_length field. On output, PMDF_send will write the status of that address back to the address string overwriting the address stored in that string. (The original address will be given in the text of the status message.) Also on output, the magnitude of the value contained in the item_length field will contain the length of the status message. If the value contained in the item_length field is positive, then the address was legal; if the value is negative then the address was illegal. See Section 2.5.3 for example programs which use this feature.

For each address built with PMDF_FAX_TO, PMDF_FAX_CC, PMDF_FAX_BCC, PMDF_PRT_TO, PMDF_PRT_CC, PMDF_PRT_BCC the address of a string of length at least ALFA_SIZE bytes must be specified in the item_address field. On output, the success or error message associated with the address will be returned in this string. The value in the item_length field will give the length of the status message as well as indicate if the address was legal or illegal.

The output strings will not be zero terminated.

The item_address and item_length fields are ignored by this action.

PMDF_BCC

PMDF_ENV_BCC

PMDF_HDR_BCC

Specify a Bcc: address. The item_address and item_length fields specify the address and length of a string containing a Bcc: address. The length of the address can not exceed ALFA_SIZE bytes.

PMDF_BCC is used to specify a blind carbon copy (Bcc:) address which should appear in both the message's header and envelope. PMDF_ENV_BCC is used to specify an envelope-only Bcc: address (*i.e.*, an active recipient) which should not appear in the message's header. PMDF_HDR_BCC is used to specify a header-only Bcc: address (*i.e.*, an inactive recipient) which should only appear in the message's header.

If PMDF_ADDRESS_STATUS is specified, then this string must have a maximum size of at least ALFA_SIZE bytes.

PMDF_BLANK

When processing multiple input sources, insert a blank line between the input from each source. Ordinarily, the input files are appended one after the other with no delimiters or separators. This is the action selected with the PMDF_NOBLANK item code. By specifying the PMDF_BLANK action, PMDF_send will insert a blank line between each input file. This is especially useful when the first input file is to be treated as a source of header information and the second as the

PMDF_send

message body or part thereof. This then produces the requisite blank line between the message header and body.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_CC **PMDF_ENV_CC** **PMDF_HDR_CC**

Specify a Cc: address. The `item_address` and `item_length` fields specify the address and length of a string containing a Cc: address. The length of the address can not exceed `ALFA_SIZE` bytes.

`PMDF_CC` is used to specify a carbon copy (Cc:) address which should appear in both the message's header and envelope. `PMDF_ENV_CC` is used to specify an envelope-only Cc: address (*i.e.*, an active recipient) which should not appear in the message's header. `PMDF_HDR_CC` is used to specify a header-only Cc: address (*i.e.*, an inactive recipient) which should only appear in the message's header.

If `PMDF_ADDRESS_STATUS` is specified, then this string must have a maximum size of at least `ALFA_SIZE` bytes.

PMDF_CHAIN

`PMDF_send` immediately begins processing the list of item descriptors pointed at by `item_address`. This new list will be used immediately; any remaining items in the current list will be ignored. The `item_length` field should contain the value 4, the length of a longword in bytes.

PMDF_CHANNEL

Specify the channel to act as when enqueueing the message. If not specified, then mail will be enqueued as though sent from the local, 1, channel. The `item_address` and `item_length` fields specify the address and length of a text string containing the name of the channel to act as. The length of the string can not exceed `CHANLENGTH` bytes.

PMDF_COMMENTS

Specify the body of a Comments: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Comments: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Comments: body can be specified. Additional Comments: header lines can be created with `PMDF_EXTRA_HEADER`.

PMDF_CONTENT_FILENAME **PMDF_NOCONTENT_FILENAME**

When `PMDF_CONTENT_FILENAME` is specified, the name of the message input file will be included as a parameter in the MIME Content-type: header line. This action, when specified, will hold for all subsequent input files until a `PMDF_NOCONTENT_FILENAME` action is seen in the same item list. `PMDF_NOCONTENT_FILENAME` is the default.

The `item_address` and `item_length` fields can be used to specify the file name, overriding the name of the input file.

PMDF_CONTENT_TYPE

Specify the body of a Content-type: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Content-type: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Content-type: body can be specified.

PMDF_DELIVERY_RECEIPT_TO

Specify the body of a Delivery-receipt-to: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Delivery-receipt-to: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Delivery-receipt-to: body can be specified.

PMDF_ENC_BASE64

PMDF_ENC_COMPRESSED_BASE64

Encode data from all subsequent input sources using MIME's BASE64 encoding. In the case of `PMDF_ENC_COMPRESSED_BASE64`, the data is first compressed using GNU zip.

This setting can be changed with any of the other `PMDF_ENC_item` codes. The default encoding is `PMDF_ENC_UNKNOWN`. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_BASE85

Encode data from all subsequent input sources using Adobe's ASCII85 encoding (BASE85). This setting can be changed with any of the other `PMDF_ENC_item` codes. The default encoding is `PMDF_ENC_UNKNOWN`. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_BINHEX

Encode data from all subsequent input sources using the BINHEX encoding. This setting can be changed with any of the other `PMDF_ENC_item` codes. The default encoding is `PMDF_ENC_UNKNOWN`. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_BTOA

Encode data from all subsequent input sources using the UNIX btoa encoding. This setting can be changed with any of the other `PMDF_ENC_item` codes. The default encoding is `PMDF_ENC_UNKNOWN`. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_COMPRESSED_BINARY

Compress the data with GNU zip. No other encoding of the data will be done. This setting can be changed with any of the other `PMDF_ENC_item` codes. The default encoding is `PMDF_ENC_UNKNOWN`. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_COMPRESSED_UUENCODE

PMDF_ENC_UUENCODE

Encode data from all subsequent input sources using UUENCODE. In the case of `PMDF_ENC_COMPRESSED_UUENCODE`, the data is first compressed using GNU zip.

This setting can be changed with any of the other `PMDF_ENC_item` codes. The default encoding is `PMDF_ENC_UNKNOWN`. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_HEXADECIMAL

Encode data from all subsequent input sources using a hexadecimal encoding. This setting can be changed with any of the other PMDF_ENC_item codes. The default encoding is PMDF_ENC_UNKNOWN.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_NONE

Data from all subsequent input sources is left unencoded (*i.e.*, not encoded). This setting can be changed with any of the other PMDF_ENC_item codes. The default encoding is PMDF_ENC_UNKNOWN.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_QUOTED_PRINTABLE

Encode data from all subsequent input sources using MIME's quoted printable encoding. This setting can be changed with any of the other PMDF_ENC_item codes. The default encoding is PMDF_ENC_UNKNOWN.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENC_UNKNOWN

Data from all subsequent input sources is left unencoded (*i.e.*, not encoded). This setting can be changed with any of the other PMDF_ENC_item codes. The default encoding is PMDF_ENC_UNKNOWN.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_END_LIST

Terminate an item list. This item code, when encountered, signals the end of the item list. The `item_address` and `item_length` fields are ignored by this action.

PMDF_ENV_FROM

Specify the envelope From: address to associate with a message. The `item_address` and `item_length` fields specify the address and length of a text string containing the envelope From: address to use for the message submission. The length of the string can not exceed ALFA_SIZE bytes. Only one envelope From: address can be specified.

The PMDF_ENV_FROM action should be used when the envelope From: address is not a local address. When the address is a local address, then merely the user name should be specified using the PMDF_USER action.

If this action and the PMDF_USER actions are not specified, then the user name associated with the current process will be used.

Can not be used in conjunction with the PMDF_USER or PMDF_SUB_USER item codes.

PMDF_ERRORS_TO

Specify the body of an Errors-to: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of an Errors-to: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one Errors-to: body can be specified.

PMDF_EXTRA_HEADER

Specify an additional header line to include in the message header. The `item_address` and `item_length` fields specify the address and length of the header line (field name and body) to place in the message header. The length of the string can not exceed `ALFA_SIZE` bytes. Any number of header lines can be added; use one item list entry per header line.

`PMDF_EXTRA_HEADER` is intended to be used to add header lines not supported by other item codes (*e.g.*, `PMDF_SUBJECT`, `PMDF_KEYWORDS`, *etc.*), or to specify additional instances of header lines which can multiple times (*e.g.*, Comments: header lines).

PMDF_FAX_AFTER

Specify the value to use with an `AFTER` attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_AT

Specify the value to use with an `AT` attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_AUTH

Specify the value to use with an `AUTH` attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_BCC**PMDF_FAX_CC****PMDF_FAX_TO**

Begin the specification of a FAX To:, Cc:, or Bcc: address. FAX addresses can be composed, one attribute at a time, using the `PMDF_FAX_` item codes. The attribute-value pair list is automatically assembled from the specified attribute-value pairs, properly quoted, and the domain specification appended. The actual assembly of the address is initiated when either the item list is terminated or when another `PMDF_*TO`, `PMDF_*CC`, or `PMDF_*BCC` action is encountered.

The FAX address to be built will be treated as a To: address when `PMDF_FAX_TO` is specified, as a Cc: address when `PMDF_FAX_CC` is specified, and as a Bcc: address when `PMDF_FAX_BCC` is specified.

The `PMDF_FAX_DOMAIN` and `PMDF_FAX_FN` actions must be specified for each FAX address to be assembled.

PMDF_send

The `item_address` and `item_length` fields are ignored by this action unless `PMDF_ADDRESS_STATUS` is specified in which case then the address of a string of length at least `ALFA_SIZE` bytes must be given in the `item_address` field.

PMDF_FAX_DOMAIN

Specify the domain name to associate with a FAX address which is being built up (e.g., `text-fax.example.com`). The `item_address` and `item_length` fields specify the address and length of a text string containing the domain name. The length of the string can not exceed `ALFA_SIZE` bytes.

This action must be taken when composing a FAX address with the `PMDF_FAX_` item codes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_FN

Specify the value to use with an FN attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

This action must be taken when composing a FAX address with the `PMDF_FAX_` item codes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_FSI

Specify the value to use with an FSI attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_O

Specify the value to use with an O attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_OU

Specify the value to use with an OU attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_SETUP

Specify the value to use with a SETUP attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_SFN

Specify the value to use with a SFN attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_STN

Specify the value to use with a STN attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_TN

Specify the value to use with a TN attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FAX_TTI

Specify the value to use with a TTI attribute in a FAX address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_FAX_TO`, `PMDF_FAX_CC`, or `PMDF_FAX_BCC` action must have appeared prior to using this action.

PMDF_FROM

Specify the address to use in the message header's From: header line. `item_address` and `item_length` fields specify the address and length of a text string containing the From: address. The length of the string can not exceed `ALFA_SIZE` bytes. Only one From: address can be specified.

If this action is not used, then the From: header line will be derived from the envelope From: address.

PMDF_FRUIT_OF_THE_DAY

Specify the body of a Fruit-of-the-day: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Fruit-of-the-day: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Fruit-of-the-day: body can be specified.

PMDF_HDR_ADDR

PMDF_HDR_NOADDRS

By default, `PMDF_HDR_NOADDRS`, recipient addresses must be explicitly specified and any addresses in a input header file will be ignored (but will still appear in the message header). Specify `PMDF_HDR_ADDR` to request that the message also be sent to recipient addresses found in any input header files.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_HDR_FILE

Specify the name of an input file containing message header lines. The first input file can be a file containing a message header. In this case, it should be specified using this item code rather than `PMDF_MSG_FILE`. This will ensure that the input file receives the proper processing (*e.g.*, is not encoded, accessed using text mode access, *etc.*). `PMDF_send` will use the header lines from the input file to form an initial message header. This initial header is then modified as necessary. This functionality is useful when forwarding mail.

Note that any recipient addresses in the header file will be ignored unless `PMDF_HDR_ADDR` is also specified.

The `item_address` and `item_length` fields specify the address and length of a text string containing the input file's name. The length of the string can not exceed `ALFA_SIZE` bytes.

PMDF_HDR_RESENT

PMDF_HDR_NORESENT

The `PMDF_HDR_RESENT` action selects the default behavior whereby Resent-header lines are added as necessary to the message header when the associated header line appears in any input header files. For instance, a Resent-to: header line will be added if a To: header line already appears. Specify `PMDF_HDR_NORESENT` to cause additional addresses to be added to existing header lines rather than through the introduction of Resent- header lines.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_HDR_PROC

Specify the address of a procedure which will return, one line at a time, header lines for the message header. The `item_address` field specifies the address of the procedure to invoke. `item_length` must be set to 4, the length in bytes of a longword.

The calling format which must be used by the procedure is given in Section 2.1.3.

PMDF_HDRMSG_FILE

Specify the name of an input file containing both the message header and message body. The content of the file represents an RFC 822 formatted message with at least one blank line separating the RFC 822 header from the message body.

PMDF_send will use the header lines from the input file to form an initial message header. This initial header is then modified as necessary.

The `item_address` and `item_length` fields specify the address and length of a text string containing the input file's name. The length of the string can not exceed `ALFA_SIZE` bytes.

PMDF_HDRMSG_PROC

Specify the address of a procedure which will return, one line at a time, each line of an RFC 822 formatted message. The RFC 822 header must come first, followed by at least one blank line, followed by the message body. The `item_address` field specifies the address of the procedure to invoke. `item_length` must be set to 4, the length in bytes of a longword.

The calling format which must be used by the procedure is given in Section 2.1.3.

PMDF_IGNORE_ERRORS

Send the message as long as at least one To: address was okay and at least one input source was okay. By default, the message will not be sent if any of the To: addresses are illegal (*e.g.*, bad syntax, restricted, unknown host, *etc.*) or if any of the input sources proved to be bad (*e.g.*, could not open an input file).

The `item_address` and `item_length` fields are ignored by this action.

PMDF_IMPORTANCE

Specify the body of an Importance: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of an Importance: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Importance: body can be specified.

PMDF_IS_CHANNEL

Ignore user-to-channel access checks when enqueueing mail. This should, in general, be used only by programs which do not enqueue mail in behalf of a user.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_IS_NOT_CHANNEL

Do not ignore user-to-channel access checks when enqueueing mail. This should, in general, be used by programs such as user agents which enqueue mail for users.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_KEYWORDS

Specify the body of a Keywords: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Keywords: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Keywords: body can be specified.

PMDF_MAX_BLOCKS

Specify the maximum number of blocks per message. If, when the message is enqueued, the message size exceeds this limit, then the message will be fragmented into smaller messages, each fragment no larger than the specified block size. The individual fragments are MIME compliant messages which use MIME's message/partial content type. MIME compliant mailers or user agents which receive the fragments can automatically re-assemble the fragmented message.

PMDF_send

(PMDF channels must be marked with the `defragment` keyword in order for automatic message re-assembly to occur.)

The size of a block can vary from site to site — sites can change this value from its default value of 1,024 bytes. Use the PMDF API routine `PMDF_get_block_size` to determine the size in bytes of a block. Or, alternatively, use the `PMDF_MAX_BYTES` item code instead.

The `item_address` field specifies the address of a longword integer whose value is the maximum block size per message or message fragment. `item_length` must be set to 4, the length in bytes of a longword integer.

By default, no limit is imposed. This default can be re-instated by specifying a value of -1. This limit can be simultaneously imposed with other limits.

PMDF_MAX_BYTES

Specify the maximum number of bytes per message. If, when the message is enqueued, the message size exceeds this limit, then the message will be fragmented into smaller messages, each fragment no larger than the specified byte size. The individual fragments are MIME compliant messages which use MIME's message/partial content type. MIME compliant mailers or user agents which receive the fragments can automatically re-assemble the fragmented message. (PMDF channels must be marked with the `defragment` keyword in order for automatic message re-assembly to occur.)

The `item_address` field specifies the address of a longword integer whose value is the maximum bytes per message or message fragment. `item_length` must be set to 4, the length in bytes of a longword integer.

By default, no limit is imposed. This default can be re-instated by using a value of -1. This limit can be simultaneously imposed with other limits.

PMDF_MAX_LINES

Specify the maximum number of message lines per message. If, when the message is enqueued, the number of message lines exceeds this limit, then the message will be fragmented into smaller messages, each fragment with no more than the specified number of lines. The individual fragments are MIME compliant messages which use MIME's message/partial content type. MIME compliant mailers or user agents which receive the fragments can automatically re-assemble the fragmented message. (PMDF channels must be marked with the `defragment` keyword in order for automatic message re-assembly to occur.)

The `item_address` field specifies the address of a longword integer whose value is the maximum number of message lines per message or message fragment. `item_length` must be set to 4, the length in bytes of a longword integer.

By default, no limit is imposed. This default can be re-instated by using a value of -1. This limit can be simultaneously imposed with other limits.

PMDF_MAX_TO

Specify the maximum number of envelope To: addresses per message copy. If, when the message is enqueued, the number of envelope To: addresses for the

message exceeds this limit, then the message will be broken into multiple copies, each copy with no more than the specified number of envelope To: addresses.

The `item_address` field specifies the address of a longword integer whose value is the maximum number of envelope To: addresses per message copy. `item_length` must be set to 4, the length in bytes of a longword integer.

By default, no limit is imposed. This default can be re-instated by using a value of -1. This limit can be simultaneously imposed with other limits.

PMDF_MODE_BLOCK

Access subsequent input files using block mode I/O. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`. This access mode will not be applied to input procedures.

On OpenVMS systems, this setting should be used for binary files.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MODE_RECORD

Access subsequent input sources using record mode I/O. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MODE_RECORD_CRATTRIBUTE

Access subsequent input sources using record mode I/O preserving carriage return record terminators. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MODE_RECORD_CRLFATTRIBUTE

Access subsequent input sources using record mode I/O preserving carriage return, line feed record terminators. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MODE_RECORD_LFATTRIBUTE

Access subsequent input sources using record mode I/O preserving line feed record terminators. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MODE_TEXT

Access subsequent input sources using text mode I/O. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`.

This setting or that selected with `PMDF_MODE_UNKNOWN` must be use for input files containing message header information and should be used for ordinary text files.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MODE_UNKNOWN

Access subsequent input sources using text mode I/O. This setting can be changed with any of the other `PMDF_MODE_` item codes. The default access mode is that selected with `PMDF_MODE_UNKNOWN`.

This setting or that selected with `PMDF_MODE_TEXT` must be use for input sources containing message header information and should be used for ordinary text files.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_MSG_FILE

Specify an input file to read and include in the message body. The file will be read using the current access mode and encoded using the current encoding as specified by `PMDF_MODE_` and `PMDF_ENC_` item codes.

The `item_address` and `item_length` fields specify the address and length of a text string containing the name of the input file. The length of the string can not exceed `ALFA_SIZE` bytes.

PMDF_MSG_PROC

Specify the address of a procedure which will return, one line at a time, data for the message body. Each line of input obtained from the procedure will be treated using the current access mode and encoded using the current encoding as specified by `PMDF_MODE_` and `PMDF_ENC_` item codes. Note, however, that the block access mode will not be applied to input procedures.

The `item_address` field specifies the address of the procedure to invoke. `item_length` must be set to 4, the length in bytes of a longword integer.

The calling format which must be used by the procedure is given in Section 2.1.3.

PMDF_NOADDRESS_ERRORS

Do not return status messages for To:, Cc:, and Bcc: addresses. This is the default setting. The strings containing To:, Cc:, and Bcc: addresses specified with `PMDF_TO`, `PMDF_CC`, `PMDF_BCC`, `PMDF_ENV_TO`, `PMDF_ENV_CC`, `PMDF_ENV_BCC`, `PMDF_HDR_TO`, `PMDF_HDR_CC`, and `PMDF_HDR_BCC` need only be long enough to contain the actual addresses.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_NOBLANK

When processing multiple input source, do not insert a blank line between the input from one source and the next. This is the default behavior: the input from each input source is appended one after the other with no delimiters or separators marking the transition between sources.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_NOIGNORE_ERRORS

Send the message only if all To: addresses are okay and all input sources are okay. This is the default.

The `item_address` and `item_length` fields are ignored by this action.

PMDF_ORGANIZATION

Specify the body of an Organization: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of an Organization: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one Organization: body can be specified.

PMDF_PRIORITY

Specify the body of a Priority: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Priority: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one Priority: body can be specified.

PMDF_PRIV_DISABLE_PROC

The address of a procedure to invoke immediately after enqueueing a message so as to disable process privileges. See the description of the PMDF_PRIV_ENABLE_PROC item code for details on the use of this item code.

This item code must be used in conjunction with the PMDF_PRIV_ENABLE_PROC item code.

The `item_length` field is ignored by this action.

PMDF_PRIV_ENABLE_PROC

The address of a procedure to invoke immediately before enqueueing a message so as to enable process privileges.

Privileges are required to enqueue messages. It is possible to provide PMDF_send with the address of two procedures to call. One procedure is called immediately prior to enqueueing a message thereby allowing process privileges to be enabled. The second procedure is then called immediately after the message has been enqueued thereby allowing process privileges to be disabled. See Section 2.3 for further details on the use of this item code.

This item code must be used in conjunction with the PMDF_PRIV_DISABLE_PROC item code.

The `item_length` field is ignored by this action.

PMDF_PRT_AT

Specify the value to use with an AT attribute in a printer To: address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_PRT_TO`, `PMDF_PRT_CC`, or `PMDF_PRT_BCC` action must have appeared prior to using this action.

PMDF_PRT_BCC

PMDF_PRT_CC

PMDF_PRT_CC

Begin the specification of a printer To:, Cc:, or Bcc: address. Printer addresses can be composed, one attribute at a time, using the `PMDF_PRT_item` codes. The attribute-value pair list is automatically assembled from the specified attribute-value pairs, properly quoted, and the domain specification appended. The actual assembly of the address is initiated when either the item list is terminated or when another `PMDF_*TO`, `PMDF_*CC`, or `PMDF_*BCC` action is encountered.

The printer address to be built will be treated as a To: address when `PMDF_PRT_TO` is specified, as a Cc: address when `PMDF_PRT_CC` is specified, and as a Bcc: address when `PMDF_PRT_BCC` is specified.

The `PMDF_PRT_DOMAIN` action must be specified for each printer address to be assembled.

The `item_address` and `item_length` fields are ignored by this action unless `PMDF_ADDRESS_STATUS` is specified in which case then the address of a string of length at least `ALFA_SIZE` bytes must be given in the `item_address` field.

PMDF_PRT_DOMAIN

Specify the domain name to associate with a printer address which is being built up (e.g., `printer.example.com`). The `item_address` and `item_length` fields specify the address and length of a text string containing the domain name. The length of the string can not exceed `ALFA_SIZE` bytes.

This action must be taken when composing a printer address with the `PMDF_PRT_item` codes.

A `PMDF_PRT_TO`, `PMDF_PRT_CC`, or `PMDF_PRT_BCC` action must have appeared prior to using this action.

PMDF_PRT_MS

Specify the value to use with an MS attribute in a printer To: address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed `ALFA_SIZE` bytes.

A `PMDF_PRT_TO`, `PMDF_PRT_CC`, or `PMDF_PRT_BCC` action must have appeared prior to using this action.

PMDF_PRT_O

Specify the value to use with an O attribute in a printer To: address which is being built up. The `item_address` and `item_length` fields specify the address

and length of a text string containing the value to use. The length of the string can not exceed ALFA_SIZE bytes.

A PMDF_PRT_TO, PMDF_PRT_CC, or PMDF_PRT_BCC action must have appeared prior to using this action.

PMDF_PRT_OU

Specify the value to use with an OU attribute in a printer To: address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed ALFA_SIZE bytes.

A PMDF_PRT_TO, PMDF_PRT_CC, or PMDF_PRT_BCC action must have appeared prior to using this action.

PMDF_PRT_P1

...

PMDF_PRT_P8

Specify the value to use with a P1, P2, P3, P4, P5, P6, P7, or P8 attribute in a printer To: address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed ALFA_SIZE bytes.

A PMDF_PRT_TO, PMDF_PRT_CC, or PMDF_PRT_BCC action must have appeared prior to using this action.

PMDF_PRT_TN

Specify the value to use with a TN attribute in a printer To: address which is being built up. The `item_address` and `item_length` fields specify the address and length of a text string containing the value to use. The length of the string can not exceed ALFA_SIZE bytes.

A PMDF_PRT_TO, PMDF_PRT_CC, or PMDF_PRT_BCC action must have appeared prior to using this action.

PMDF_READ_RECEIPT_TO

Specify the body of a Read-receipt-to: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Read-receipt-to: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one Read-receipt-to: body can be specified.

PMDF_REFERENCES

Specify the body of a References: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a References: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one References: body can be specified.

PMDF_REPLY_TO

Specify the body of a Reply-to: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Reply-to: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one Reply-to: body can be specified.

PMDF_RESENT_FROM

Specify the body of a Resent-From: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Reply-to: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Reply-to: body can be specified.

PMDF_RESENT_REPLY_TO

Specify the body of a Resent-reply-to: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Resent-reply-to: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Resent-reply-to: body can be specified.

PMDF_SENSITIVITY

Specify the body of a Sensitivity: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Sensitivity: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Sensitivity: body can be specified.

PMDF_SUBADDRESS

Specify a subaddress to use when generating a return address from a user name specified with the `PMDF_USER` item code. The `item_address` and `item_length` fields specify the address and length of a text string containing the subaddress. The length of the string can not exceed `ALFA_SIZE` bytes. Only one subaddress can be specified per message.

The `PMDF_USER` action must be used in conjunction with this item code.

PMDF_SUBJECT

Specify the body of a Subject: header line. The `item_address` and `item_length` fields specify the address and length of a text string to place in the body of a Subject: header line. The length of the string can not exceed `ALFA_SIZE` bytes. Only one Subject: body can be specified.

PMDF_TO

PMDF_ENV_TO

PMDF_HDR_TO

Specify a To: address. The `item_address` and `item_length` fields specify the address and length of a string containing a To: address. The length of the address can not exceed `ALFA_SIZE` bytes.

`PMDF_TO` is used to specify a To: address which should appear in both the message's header and envelope. `PMDF_ENV_TO` is used to specify an envelope-only To: address (*i.e.*, an active recipient) which should not appear in the message's header. `PMDF_HDR_TO` is used to specify a header-only address To: (*i.e.*, an inactive recipient) which should only appear in the message's header.

If `PMDF_ADDRESS_STATUS` is specified, then this string must have a maximum size of at least `ALFA_SIZE` bytes.

PMDF_USER

Specify the user name to use for the envelope From: and header line From: addresses. The `item_address` and `item_length` fields specify the address and length of a text string containing the user name.

The `PMDF_ENV_FROM` action should be used when the envelope From: address

is not a local address. When the address is a local address, then merely the user name should be specified using the PMDF_USER action.

If this action and the PMDF_ENV_FROM actions are not specified, then the user name associated with the current process will be used.

Under OpenVMS, WORLD privilege — as a default privilege — is required to use this action when the specified From: address does not agree with the user name of the process enqueueing the message. On UNIX, the process must have the same (real) UID as the root or pmdf account. If the process lacks sufficient privileges, the SS\$_NOWORLD (OpenVMS) or PMDF__INSUFPRIV (UNIX) error will be returned. On NT systems, the process must be a privileged process such as Administrator.

Can not be used in conjunction with the PMDF_ENV_FROM item code.

PMDF_WARNINGS_TO

Specify the body of a Warnings-to: header line. The item_address and item_length fields specify the address and length of a text string to place in the body of a Warnings-to: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one Warnings-to: body can be specified.

PMDF_X_ORGANIZATION

Specify the body of a X-Organization: header line. The item_address and item_length fields specify the address and length of a text string to place in the body of a X-Organization: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one X-Organization: body can be specified.

PMDF_X_PS_QUALIFIERS

Specify the body of a X-PS-Qualifiers: header line. The item_address and item_length fields specify the address and length of a text string to place in the body of a X-PS-Qualifiers: header line. The length of the string can not exceed ALFA_SIZE bytes. Only one X-PS-Qualifiers: body can be specified.

DESCRIPTION Send a message. The processing carried out to address the message, generate the message's header and body, and enqueue the message is specified through the **item_list** argument. Refer to Section 2.1 for details on how to use PMDF_send.

In the event of an error (an even return value), no message will be sent.

RETURN VALUES

PMDF__OK	Normal, successful completion.
PMDF__ADDRERRS	One or more illegal envelope To: addresses prevented the message from being sent.
PMDF__ALLADDRBAD	Message contained no legal envelope To: addresses; no message sent.
PMDF__BADITEMADDR	item_address associated with an item list entry is illegal.
PMDF__BADITEMCODE	Unrecognized item_code specified in an item list entry.
PMDF__BADITEMSIZE	item_length associated with an item list entry is incorrect.

Callable SEND

PMDF_send

PMDF__ERRFDLPROC	An error occurred while attempting to process an OpenVMS file descriptor for an input file.
PMDF__ERROPENINP	An error occurred while attempting to open an input file.
PMDF__ERRPROCINP	An error occurred while processing an input source.
PMDF__FCRT	File create error. The message could not be placed in the PMDF message queues. This is typically due to insufficient privileges although other possibilities exist such as insufficient disk space. Message not enqueued.
PMDF__FILOPNERRS	An error occurred while processing an input source.
PMDF__FOPN	Initialization failed. One or more PMDF configuration files could not be accessed. PMDF configuration files are incorrectly protected.
PMDF__HOST	Illegal address specified (e.g., bad syntax, illegal mail box name, corresponds to a restricted mailing list, etc.).
PMDF__INCOMPITMS	Incompatible item codes specified.
PMDF__INSUFPRIV	Process must have the same (real) UID as either the <code>root</code> or <code>pmdf</code> account in order to specify with the <code>PMDF_USER</code> item code a user name different than that of the current process. This error code is only returned on UNIX.
PMDF__MISGNSTART	A <code>PMDF_FAX_*</code> or <code>PMDF_PRT_*</code> item code was used without first specifying a <code>PMDF_FAX_TO</code> , <code>PMDF_FAX_CC</code> , <code>PMDF_FAX_BCC</code> , <code>PMDF_PRT_TO</code> , <code>PMDF_PRT_CC</code> , or <code>PMDF_PRT_BCC</code> item code.
PMDF__NO	Initialization failed owing to a version mismatch between the current version of PMDF and the site's compiled configuration. Either the PMDF configuration needs to be recompiled or the character set tables need to be recompiled.
PMDF__NOADDRESSES	No To:, Cc:, or Bcc: addresses specified.
PMDF__NOOP	Message had no envelope To: addresses; its delivery was effected by simply deleting it.
PMDF__STRTRUERR	A string specified in one of the item list entries exceeds, in length, the maximum size allowed for the associated item code.
SS\$_NOWORLD	OpenVMS WORLD default privilege is required to specify with the <code>PMDF_USER</code> item code a user name different than that of the current process. This error code is only returned on OpenVMS.
On OpenVMS Systems	Any error returned by the <code>\$GETJPI</code> System Service or the <code>STR\$TRIM</code> or <code>STR\$UPCASE</code> OpenVMS Run Time Library routines.

A Error Codes

Each of the error codes returned by the API are described below. Note that the codes returned by the API follow the OpenVMS convention of success codes having an odd value and error codes having an even value. Thus, programs can test the low bit of a return value to see if an error occurred.

PMDF__ADDRERRS

One or more illegal envelope "To:" addresses prevented a message from being sent with `PMDF_send`.

PMDF__ALLADDRBAD

A message to be sent with `PMDF_send` contained no legal "To:", "Cc:", or "Bcc:" addresses.

PMDF__BAD

Bad parameter value supplied. An illegal value for the **property** parameter to the address property routines was specified, or an illegal value for the **database** parameter to the database routines was specified.

PMDF__BADCONTEXT

An bad context variable was specified.

PMDF__BADITEMADDR

An illegal `ITEM_ADDRESS` was present in an item list passed to `PMDF_send`.

PMDF__BADITEMCODE

An illegal (undefined) `ITEM_CODE` was specified in an item list passed to `PMDF_send`.

PMDF__BADITEMSIZE

An illegal `ITEM_LENGTH` was specified in an item list passed `PMDF_send`.

PMDF__CANOPNDAT

The specified database could not be opened or does not exist. If it does exist, then it can be incorrectly protected or formatted.

PMDF__CANTUPDAT

An attempt to update the database failed. That is, an attempt to add or remove an entry failed. It can be that the database doesn't exist or is incorrectly protected or formatted. In the case of a failed entry addition, it can be a disk quota problem or lack of free disk space.

PMDF__DONE

`PMDF_DONE` is actually a success code and not an error. It is returned by `PMDFoptionRead` to indicate that no option file existed.

PMDF__DUPENTRY

Entry could not be added to the database as it would otherwise duplicate an existing entry. Specify a value of true for the **replace** argument to `PMDFdatabaseAddEntry` in order to override the existing entry.

Error Codes

PMDF__ENTWONFIT

Entry is too long to fit in the specified database. See the description of `PMDFdatabaseAddEntry` for a discussion of maximum database entry lengths.

PMDF__EOF

The interpretation of this error code depends upon which dequeue processing routine returned it.

- `PMDFgetMessage`: a `PMDF__EOF` indicates that there are no more messages to process.
- `PMDFgetRecipient`: the entire envelope "To:" address list has been read.
- `PMDFreadLine`: the end of the message has been reached; there are no more lines to be read from this message.
- `PMDFreadText`: the end of the message has been reached; there are no more lines to be read from this message.

PMDF__ERRFDLPROC

`PMDF_send` encountered an error while attempting to process an OpenVMS file descriptor for an input file.

PMDF__ERROPENINP

`PMDF_send` was unable to open an input file.

PMDF__ERRPROCINP

`PMDF_send` encountered an error while processing an input source.

PMDF__FATERRLIB

A call to `LIB$SCOPY_R_DX` failed owing to a fatal internal error in the OpenVMS Run Time Library. This has prevented the API from writing data into a string passed by descriptor to an API routine. Consult the description of the particular routine returning this error in order to determine what processing, if any, was accomplished.

PMDF__FCRT

`PMDFenqueueMessage` or `PMDF_send` was unable to create a message file in the message queue directories. Usually, this means that the process lacks sufficient privileges to create a file in the PMDF message queues. However, it can indicate other problems (*e.g.*, disk full, quota exceeded,*etc.*).

PMDF__FILOPNERRS

`PMDF_send` encountered an error while attempting to process an input source.

PMDF__FOPN

`PMDFinitialize` or `PMDF_send` was unable to load PMDF configuration information. One or more PMDF configuration files could not be accessed. This usually means that one or more PMDF configuration files are incorrectly protected; however, it can also be caused by missing or corrupted files.

PMDF__HEANOTKNW

An unknown header line type was specified to `PMDFaddHeaderLine` or `PMDFdeleteHeaderLine`. To proceed with the operation anyhow, recall the procedure specifying `HL_OTHER` as the header line type.

PMDF__HOST

An illegal or restricted address was passed to `PMDFaddRecipient` or `PMDF_send`. In the case of `PMDFaddRecipient`, call `PMDFgetErrorText` to determine the nature of the problem; in the case of `PMDF_send`, the error, on a per address basis, will be described in the string associated with each address by a `PMDF_ERROR_TEXT` item code. In either case, the text of the error will be one of the following:

- `Unknown host or domain`: the address references a host or domain which is not recognized by the site's `PMDF` configuration.
- `List is currently reserved and locked`: the address is for a mailing list which is currently locked and cannot be used.
- `You are not allowed to use this list`: the address is for a restricted mailing list which does not accept postings from the specified "From:" address.
- `No addressees in`: the address translates to an empty address or address list.
- `Channel size limit exceeded`: message size exceeds size limit imposed one or more destination channels. This limit was imposed by the postmaster and set with a channel keyword.
- `Channel line limit exceeded`: message size exceeds line count limit imposed one or more destination channels. This limit was imposed by the postmaster and set with a channel keyword.
- `You are not allowed to use this address`: the combination of source channel, "From:" address, destination channel, and "To:" address is not permitted by site imposed access restrictions.

PMDF__INCOMPITMS

`PMDF_send` was passed an item list containing incompatible item codes.

PMDF__INSUFPRIV

Calling process must have the same (real) UID as either the `root` or `pmdf` account in order to specify with the `PMDF_USER` item code a user name different than that of the current process.

PMDF__INSVIRMEM

A call to `LIB$GET_VM` made by `LIB$SCOPY_R_DX` has failed owing to insufficient virtual memory. This has prevented the API from writing data into a string passed by descriptor to an API routine. Consult the description of the particular routine returning this error in order to determine what processing, if any, was accomplished. The process probably needs to have its page file quota increased or the system's virtual page count can need to be increased.

PMDF__INVSTRDES

An invalid string descriptor was passed to an API routine. The API routines require that all string descriptors be passed by reference.

PMDF__MISGNSTART

In an item list passed to `PMDF_send`, a `PMDF_FAX_*` or `PMDF_PRT_*` item code was used without first specifying a `PMDF_FAX_TO`, `PMDF_FAX_CC`, `PMDF_FAX_BCC`, `PMDF_PRT_TO`, `PMDF_PRT_CC`, or `PMDF_PRT_BCC` item code to start a FAX or printer address specification.

Error Codes

PMDF__NAUTH

An address passed to `PMDFaddRecipient` can not be used by the sending address — it is a restricted address or mailing list. Further information can be obtained by calling `PMDFgetErrorText`.

PMDF__NO

The interpretation of this error code depends upon which routine returned it.

- `PMDFaddressGet`, `PMDFaddressGetProperty`: value for the **index** parameter was out of range.
- `PMDFenqueueMessage`: a temporary processing error occurred; the message enqueue was not successful.
- `PMDFgetAddressProperty`: specified address contained more than one address. Use `PMDFaddressParseLine` and `PMDFaddressGetProperty` instead.
- `PMDFgetRecipient`: the message file was corrupt and should be deleted by calling `PMDFdequeueMessage`.
- `PMDFinitialize`: the site's PMDF configuration file needs to be recompiled with `pmdf cnbuild` or the site's character set tables need to be recompiled with `pmdf chbuild`. *OpenVMS only*: After recompiling either set of tables, they need to be reinstalled.
- `PMDFgetChannelStats`: After ten attempts, each one second apart, `PMDFgetChannelStats` was unable to obtain a lock on the channel statistics cache.
- `PMDFmappingLoad`: `PMDFinitialize` has not yet been called. PMDF must be initialized before loading any mapping tables.
- `PMDFrewindMessage`: there is an inconsistency in the message file.
- `PMDF_send`: same as `PMDFinitialize`.
- `PMDFstartMessageEnvelope`: there is an error in the site's PMDF configuration. Either the specified channel does not exist or there is an error in the PMDF configuration file.

PMDF__NOOP

A message enqueued with `PMDFenqueueMessage` or `PMDF_send` had no envelope "To:" addresses and was therefore simply deleted.

PMDF__NOADDRESSES

An item list passed to `PMDF_send` contained no "To:", "Cc:", or "Bcc:" addresses.

PMDF__NOCHANNEL

Either the channel name to associate with the executing program could not be determined, or once determined the channel could not be located in the PMDF configuration file. On OpenVMS systems, the channel name is generally specified with the `PMDF_CHANNEL` logical which should translate to the name of the channel to use.

PMDF__NOMAPPING

The specified mapping table could not be loaded. Check to see that the mapping file exists. If it does exist, check to ensure that the mapping table name is correct.

PMDF__OK

Successful, normal completion.

PMDF__PARSE

An address passed to `PMDFaddRecipient` had bad or otherwise illegal syntax. An address passed to `PMDFgetAddressProperty` contained no legal addresses (*i.e.*, either contained no addresses or had one or more syntactically illegal addresses).

PMDF__STRTRU

A string passed to an API routine was not large enough. The data written to this string by an API routine was truncated to fit. Depending upon the application, the truncated data can or cannot be usable.

PMDF__STRTRUERR

A string passed to an API routine was not large enough and truncating the data to be written to the string would only result in an error (*i.e.*, the data is not usable when truncated).

PMDF__USER

A bad or illegal user name was specified in a local address passed to `PMDFaddRecipient`.

SS\$_NOWORLD

OpenVMS WORLD default privilege required to specify with the `PMDF_USER` item code a user name different than that of the current process.

Glossary

Channel program: Loosely speaking, any program which enqueues or dequeues messages to or from PMDF's message queues.

Dequeue: The act of removing a mail message from PMDF's message queues.

Enqueue: The act of submitting for transmission a mail message to PMDF.

Envelope: The message's transport layer To: and From: addressing information is contained in the message envelope.

Master channel program: Any program which enqueues messages to PMDF's message queues.

MIME: See RFC 2045–2049.

MTA: Message transfer agent; *e.g.*, PMDF.

RFC: Request For Comments; the Internet's method of publishing documents.

RFC 822: RFC 822, written by David Crocker, is the Internet standards document entitled *Standard for the Format of ARPA Internet Text Messages*. Messages in PMDF's message queues conform to this standard; *i.e.*, RFC 822 is the format which PMDF uses internally.

RFC 1123: RFC 1123, edited by Robert Braden, is the Internet standards document entitled *Internet Host Requirements — Application and Support*. PMDF adheres to the requirements put forth by this document.

RFCs 2045–2049: RFCs 2045–2049, commonly referred to as MIME, written by Nathaniel Borenstein and Ned Freed, are the Internet standards track documents describing the format of Internet message bodies. PMDF uses the specifications laid out in this document when forming multipart messages, encoded messages, *etc.* Note that RFCs 2045–2049 replaced RFCs 1521–1522 and 1431, previous drafts of MIME.

RFC 1566: RFC 1566, sometimes referred to as MADMAN, written by Steve Kille and Ned Freed, is the Internet standards track protocol entitled *Mail Monitoring MIB*. PMDF accumulates the necessary message traffic statistics needed for this MIB. The concept of “group” used in the MIB is identified with a PMDF channel. The `PMDF.getChannelStats` routine can be used to access the messages traffic statistics, referred to as channel statistics.

RFCs 1891–1894: RFCs 1891–1894, sometimes referred to as NOTARY, written by Keith Moore and Greg Vaudreuil, are the Internet standards track documents for the format and handling of notification messages.

Glossary

Slave channel program: Any program which dequeues messages from PMDF's message queues.

UA: User agent; *e.g.*, the VMS MAIL utility.

Index

A

Aborting

- dequeue • 1–85
- enqueue • 1–42

Accessing messages

- See dequeuing messages

Addresses

- Bcc: • 2–23
- Cc: • 2–24
- envelope
 - See envelope, message
- From: • 2–2
- header
 - See header, message
- To: • 2–38
- To:, Cc:, and Bcc: • 2–2

Address parsing • 1–50, 1–51, 1–53, 1–55, 1–93

ALFA_SIZE = 252 bytes • 1–41

Aliases, inhibiting • 1–57

apidef.h

- OpenVMS: PMDF_COM:apidef.h • 1–8
- UNIX, NT: /pmdf/include/apidef.h • 1–8

apidef.pen

- OpenVMS: PMDF_EXE:apidef.pen • 1–8

B

Bcc: addresses • 1–47, 2–23

BIGALFA_SIZE = 1024 bytes • 1–41

Block size • 1–96

Body, message

- description • 1–2
- enqueueing • 1–183
- PMDF_send • 2–3
- starting • 1–183

Bouncing messages • 1–161

- example program • 1–30

C

Calling dependencies • 1–38

Cc: addresses • 1–47, 2–24

CHANLENGTH = 32 bytes • 1–41

Channel counters • 1–97

Channel keywords

- defragment • 1–173
- determining which are set • 1–103
- headerbottom • 1–103
- headerinc • 1–103
- headeromit • 1–103
- logging • 1–12
- master_debug • 1–103
- slave_debug • 1–103

Channel log file • 1–12

Channel name • 1–4, 1–102

Comments: header line • 2–24

Compiling programs • 1–13, 2–5

Content-type: header line • 2–24, 2–25

Counters, channels • 1–97

D

DATA_LENGTH = 80 bytes • 1–41

Date • 1–105

Date: header line • 1–2, 1–187

Debugging • 1–12, 1–73

Debug output • 1–12, 1–130

Deferring queued messages • 1–6, 1–85

defragment keyword • 1–173, 2–32

Deleting a message

- dequeue • 1–85
- enqueue • 1–42

Delivery failure log • 1–86

- reading • 1–151
- writing • 1–85

Delivery receipts • 1–181

Delivery-receipt-to: header line • 1–158, 1–181, 2–25

Dequeuing messages • 1–4 to 1–7

- aborting • 1–6, 1–85
- accessing a message • 1–113
- basic steps • 1–5
- bouncing messages • 1–161
- contexts • 1–7
- copying a message • 1–61
- debugging • 1–12, 1–73
- deferring • 1–85
- ending • 1–82
- example • 1–16, 1–23, 1–30
- logging • 1–12
- message locking • 1–6

Index

- Dequeuing messages (cont'd)
 - privileges required • 1–12
 - reading • 1–154, 1–156
 - re-reading messages • 1–167
 - returning messages • 1–30, 1–161
 - rewinding messages • 1–167

E

- Enqueueing messages
 - To:, Cc:, and Bcc: addresses • 1–179
- Enqueuing messages • 1–2 to 1–4
 - aborting • 1–42
 - basic steps • 1–3
 - callable SEND • 2–1
 - contexts • 1–7
 - copying a message • 1–61
 - debugging • 1–12, 1–73
 - delivery receipts • 1–158, 1–181
 - example • 1–14, 1–23
 - fragmenting • 1–172
 - inhibiting aliases • 1–57
 - killing • 1–42
 - logging • 1–12
 - message body • 1–183
 - PMDF_send • 2–1
 - privileges required • 1–12
 - read receipts • 1–158, 1–181
 - receipts • 1–158, 1–181
 - simple example
 - PMDF_send • 2–5
 - size limits • 1–172
 - starting • 1–90
 - submitting • 1–91
 - writing message lines • 1–191, 1–195
- Envelope, message
 - description • 1–2
 - envelope id • 1–107, 1–170
 - From: address • 1–2, 1–184
 - PMDF_send • 2–2, 2–38
 - NOTARY flags • 1–122, 1–177
 - To: addresses • 1–2
 - PMDF_send • 2–2, 2–38
 - reading • 1–119
 - writing • 1–47
- Envelope id • 1–107, 1–170
- Environment files
 - See files
- Error codes
 - PMDF__ADDRRRS • A–1
 - PMDF__ALLADDRBAD • A–1
 - PMDF__BAD • A–1
 - PMDF__BADCONTEXT • A–1
 - PMDF__BADITEMADDR • A–1

- Error codes (cont'd)
 - PMDF__BADITEMCODE • A–1
 - PMDF__BADITEMSIZE • A–1
 - PMDF__CANOPNDAT • A–1
 - PMDF__CANTUPDAT • A–1
 - PMDF__DONE • A–1
 - PMDF__DUPENTRY • A–1
 - PMDF__ENTWONFIT • A–1
 - PMDF__EOF • A–2
 - PMDF__ERRFDLPROC • A–2
 - PMDF__ERROPENINP • A–2
 - PMDF__ERRPROCINP • A–2
 - PMDF__FATERRLIB • A–2
 - PMDF__FCRT • A–2
 - PMDF__FILOPNERRS • A–2
 - PMDF__FOPN • A–2
 - PMDF__HEANOTKNW • A–2
 - PMDF__HOST • A–2
 - PMDF__INCOMPITMS • A–3
 - PMDF__INSUFPRIV • A–3
 - PMDF__INSVIRMEM • A–3
 - PMDF__INVSTRDES • A–3
 - PMDF__MISGNSTART • A–3
 - PMDF__NAUTH • A–3
 - PMDF__NO • A–4
 - PMDF__NOADDRESSES • A–4
 - PMDF__NOCHANNEL • A–4
 - PMDF__NOMAPPING • A–4
 - PMDF__NOOP • A–4
 - PMDF__OK • A–4
 - PMDF__PARSE • A–4
 - PMDF__STRTRU • A–5
 - PMDF__STRTRUERR • A–5
 - PMDF__USER • A–5
 - SS\$_NOWORLD • A–5
- Errors
 - during channel processing • 1–6
 - obtaining information about • 1–109
- Errors-to: header line • 2–26
- Examples • 1–13 to 1–35, 2–5 to 2–17
 - dequeuing & re-enqueuing messages • 1–23
 - dequeuing & returning messages • 1–30
 - dequeuing messages • 1–16
 - enqueuing messages • 1–14
 - PMDF_send
 - enqueuing messages • 2–5
 - FAX addresses • 2–10
 - initial message header • 2–7
 - input procedure • 2–15
 - multiple recipients • 2–10

F

Failure log
 See delivery failure log

Files

- apidef.h
 - OpenVMS: PMDF_COM:apidef.h • 1-8
 - UNIX, NT: /pmdf/include/apidef.h • 1-8
- apidef.pen
 - OpenVMS: PMDF_EXE:apidef.pen • 1-8

Files, option • 1-138, 1-140, 1-142, 1-144

Foutines

- PMDFaddRecipient • 1-47

Fragmenting messages • 1-172

From: header line • 1-2, 1-188, 2-29

Fruit-of-the-day: header line • 2-30

H

Header, message

- Content-type: header line • 2-24, 2-25
- Date: header line • 1-2, 1-187
- Delivery-receipt-to: header line • 1-181, 2-25
- description • 1-2
- enqueueing • 1-186
- Errors-to: header line • 2-26
- From: address • 1-184
 - PMDF_send • 2-2, 2-29
- From: header line • 1-2, 1-188, 2-29
- Fruit-of-the-day: header line • 2-30
- Importance: header line • 2-31
- Keywords: header line • 2-31
- PMDF_send • 2-3
- Priority: header line • 2-35
- Read-receipt-to: header line • 1-181, 2-37
- References: header line • 2-37
- Reply-to: header line • 2-37
- Resent-from: header line • 2-38
- Resent-reply-to: header line • 2-38
- Sensitivity: header line • 2-38
- starting • 1-186
- Subject: header line • 1-193, 2-38
- To:, Cc:, and Bcc: addresses • 1-47
 - PMDF_send • 2-2, 2-23, 2-24, 2-38
- Warnings-to: header line • 2-39
- X-Organization: header line • 2-39
- X-PS-qualifiers: header line • 2-39

headerbottom keyword • 1-103

Header files

- See files

headerinc keyword • 1-103

headeromit keyword • 1-103

Host name • 1-111

I

I/O • 1-12, 1-130

Importance: header line • 2-31

Include files

- See files

Infinite loop

- See loop, infinite

Item code • 2-22

Item list • 2-22

item_address • 2-22

item_length • 2-22

K

Keywords: header line • 2-31

KEY_LENGTH = 32 bytes • 1-41

L

Linking programs • 1-13, 2-5

Local host name • 1-111

Locking messages • 1-6

Log file • 1-59

Log file output • 1-12, 1-130

Logging • 1-12

logging keyword • 1-12

LONG_DATA_LENGTH = 252 bytes • 1-41

LONG_KEY_LENGTH = 80 bytes • 1-41

Loop, infinite

- See infinite loop

M

Mail, sending

- See enqueueing messages

master_debug keyword • 1-103

Message body

- See body, message

Message envelope

- See envelope, message

Index

Message header

See header, message

Message id

obtaining • 1–115

MIME • Glossary–1

Multi-threaded applications • 1–174

Multithreaded applications • 1–7

Mutex • 1–7, 1–174

N

NOTARY • Glossary–1

NOTARY flags

message dequeue • 1–122

message enqueue • 1–177

O

Official local host name • 1–111

Option files, reading • 1–138, 1–140, 1–142, 1–144

Order dependencies • 1–38

Output • 1–12, 1–130

P

PMDF log file • 1–10, 1–59

PMDF_send

address status • 2–2

basic steps • 2–1

body, message • 2–3

calling • 2–21

description • 2–21

From: address • 2–2, 2–38

header, message • 2–3

input procedures • 2–3

calling format • 2–3

item codes • 2–22, 2–23

PMDF_ADDRESS_STATUS • 2–2, 2–10, 2–23

PMDF_BCC • 2–2, 2–23

PMDF_BLANK • 2–23

PMDF_CC • 2–2, 2–24

PMDF_CHAIN • 2–24

PMDF_CHANNEL • 2–24

PMDF_COMMENTS • 2–24

PMDF_CONTENT_FILENAME • 2–24

PMDF_CONTENT_TYPE • 2–25

PMDF_DELIVERY_RECEIPT_TO • 2–25

PMDF_ENC_BASE64 • 2–25

PMDF_ENC_BASE85 • 2–25

PMDF_send

item codes (cont'd)

PMDF_ENC_BINHEX • 2–25

PMDF_ENC_BTOA • 2–25

PMDF_ENC_COMPRESSED_BASE64 • 2–25

PMDF_ENC_COMPRESSED_BINARY • 2–25

PMDF_ENC_COMPRESSED_UUENCODE • 2–25

PMDF_ENC_HEXADecimal • 2–26

PMDF_ENC_NONE • 2–26

PMDF_ENC_QUOTED_PRINTABLE • 2–26

PMDF_ENC_UNKNOWN • 2–26

PMDF_ENC_UUENCODE • 2–25

PMDF_END_LIST • 2–26

PMDF_ENV_BCC • 2–2, 2–23

PMDF_ENV_CC • 2–2, 2–24

PMDF_ENV_FROM • 2–2, 2–26

PMDF_ENV_TO • 2–2, 2–38

PMDF_ERRORS_TO • 2–26

PMDF_EXPAND_LIMIT • 2–32

PMDF_EXTRA_HEADER • 2–27

PMDF_FAX_AFTER • 2–27

PMDF_FAX_AT • 2–27

PMDF_FAX_AUTH • 2–27

PMDF_FAX_BCC • 2–27

PMDF_FAX_CC • 2–27

PMDF_FAX_DOMAIN • 2–28

PMDF_FAX_FN • 2–28

PMDF_FAX_FSI • 2–28

PMDF_FAX_O • 2–28

PMDF_FAX_OU • 2–28

PMDF_FAX_SETUP • 2–29

PMDF_FAX_SFN • 2–29

PMDF_FAX_STN • 2–29

PMDF_FAX_TN • 2–29

PMDF_FAX_TO • 2–10, 2–27

PMDF_FAX_TTI • 2–29

PMDF_FROM • 2–29

PMDF_FRUIT_OF_THE_DAY • 2–30

PMDF_HDRMSG_FILE • 2–30

PMDF_HDRMSG_PROC • 2–31

PMDF_HDR_ADDRS • 2–30

PMDF_HDR_BCC • 2–2, 2–23

PMDF_HDR_CC • 2–2, 2–24

PMDF_HDR_FILE • 2–3, 2–7, 2–30

PMDF_HDR_NOADDRS • 2–30

PMDF_HDR_NORESENT • 2–30

PMDF_HDR_PROC • 2–3, 2–30

PMDF_HDR_RESENT • 2–30

PMDF_HDR_TO • 2–2, 2–38

PMDF_IGNORE_ERRORS • 2–31

PMDF_IMPORTANCE • 2–31

PMDF_IS_CHANNEL • 2–31

PMDF_IS_NOT_CHANNEL • 2–31

PMDF_KEYWORDS • 2–31

PMDF_MAX_BLOCKS • 2–31

PMDF_MAX_BYTES • 2–32

PMDF_send
 item codes (cont'd)
 PMDF_MAX_LINES • 2–32
 PMDF_MODE_BLOCK • 2–33
 PMDF_MODE_RECORD • 2–33
 PMDF_MODE_RECORD_CRATTRIBUTE • 2–33
 PMDF_MODE_RECORD_CRLFATTRIBUTE • 2–33
 PMDF_MODE_RECORD_LFATTRIBUTE • 2–33
 PMDF_MODE_TEXT • 2–34
 PMDF_MODE_UNKNOWN • 2–34
 PMDF_MSG_FILE • 2–3, 2–34
 PMDF_MSG_PROC • 2–3, 2–15, 2–34
 PMDF_NOADDRESS_ERRORS • 2–34
 PMDF_NOBLANK • 2–35
 PMDF_NOCONTENT_FILENAME • 2–24
 PMDF_NOIGNORE_ERRORS • 2–35
 PMDF_ORGANIZATION • 2–35
 PMDF_PRIORITY • 2–35
 PMDF_PRIV_DISABLE_PROC • 2–4, 2–35
 PMDF_PRIV_ENABLE_PROC • 2–4, 2–35
 PMDF_PRT_AT • 2–36
 PMDF_PRT_BCC • 2–36
 PMDF_PRT_CC • 2–36
 PMDF_PRT_DOMAIN • 2–36
 PMDF_PRT_MS • 2–36
 PMDF_PRT_O • 2–36
 PMDF_PRT_OU • 2–37
 PMDF_PRT_Pn • 2–37
 PMDF_PRT_TN • 2–37
 PMDF_PRT_TO • 2–36
 PMDF_READ_RECEIPT_TO • 2–37
 PMDF_REFERENCES • 2–37
 PMDF_REPLY_TO • 2–37
 PMDF_RESENT_FROM • 2–38
 PMDF_RESENT_REPLY_TO • 2–38
 PMDF_SENSITIVITY • 2–38
 PMDF_SUBADDRESS • 2–38
 PMDF_SUBJECT • 2–38
 PMDF_TO • 2–2, 2–38
 PMDF_USER • 2–2, 2–38
 PMDF_WARNINGS_TO • 2–39
 PMDF_X_ORGANIZATION • 2–39
 PMDF_X_PS_QUALIFIERS • 2–39
 summary • 2–17
 item descriptor fields • 2–22
 item_address • 2–22
 item_length • 2–22
 item_list argument • 2–22
 overview • 2–1
 status messages • 2–2
 To:, Cc:, and Bcc: addresses • 2–2, 2–23, 2–24, 2–38
 Postmaster address • 1–117
 Priority: header line • 2–35
 Privileges • 1–12, 2–4
 PMDF_send • 2–2
 PMDF_USER item code • 2–2

Privileges (cont'd)
 VMS WORLD • 2–2

Q

Queue cache database
 closing • 1–10, 1–60
 dumping • 1–147

R

Reading messages

 See dequeuing messages
 Read-receipt-to: header line • 1–158, 1–181, 2–37
 Receipts • 1–181
 controlling • 1–158
 delivery receipts • 1–181, 2–25
 read receipts • 1–181
 Read receipts • 2–37
 Re-entrancy • 1–7, 1–174
 References: header line • 2–37
 Reply-to: header line • 2–37
 Resent-from: header line • 2–38
 Resent-reply-to: header line • 2–38
 Returning messages • 1–161
 example program • 1–30
 Rewinding messages • 1–167
 RFC 1123 • 1–1, Glossary–1
 RFC 1566 • 1–98, Glossary–1
 RFC 1891–1894 • Glossary–1
 RFC 2045–2049 • 1–1, Glossary–1
 RFC 822 • 1–1, Glossary–1
 Routines
 order dependencies • 1–38
 PMDFabortMessage • 1–42
 PMDFabortProgram • 1–43
 PMDFaddHeaderLine • 1–45
 PMDFaddressDispose • 1–50
 PMDFaddressGet • 1–51
 PMDFaddressGetProperty • 1–53
 PMDFaddressParseList • 1–55
 PMDFaliasNoExpansion • 1–57
 PMDFcancelCallBack • 1–58
 PMDFcloseLogFile • 1–59
 PMDFcloseQueueCache • 1–60
 PMDFcopyMessage • 1–61
 PMDFdatabaseAddEntry • 1–63
 PMDFdatabaseClose • 1–67
 PMDFdatabaseDeleteEntry • 1–68
 PMDFdatabaseGetEntry • 1–70
 PMDFdebug • 1–73

Index

Routines (cont'd)

- PMDFdecodeMessage • 1–75
- PMDFdeferMessage • 1–78
- PMDFdeleteHeaderLine • 1–80
- PMDFdequeueEnd • 1–82
- PMDFdequeueInitialize • 1–83
- PMDFdequeueMessage • 1–84
- PMDFdequeueMessageEnd • 1–85
- PMDFdisposeChannelCounters • 1–87
- PMDFdisposeHeader • 1–88
- PMDFdone • 1–89
- PMDFenqueueInitialize • 1–90
- PMDFenqueueMessage • 1–91
- PMDFgetAddressProperty • 1–93
- PMDFgetBlockSize • 1–96
- PMDFgetChannelCounters • 1–97
- PMDFgetChannelName • 1–102
- PMDFgetDateTime • 1–105
- PMDFgetEnvelopeId • 1–107
- PMDFgetErrorText • 1–109
- PMDFgetHostName • 1–111
- PMDFgetMessage • 1–113
- PMDFgetMessageId • 1–115
- PMDFgetPostmasterAddress • 1–117
- PMDFgetRecipient • 1–119
- PMDFgetRecipientFlags • 1–122
- PMDFgetUniqueString • 1–124
- PMDFgetUserName • 1–126
- PMDFinitialize • 1–128
- PMDFlog • 1–130
- PMDFmappingApply • 1–132
- PMDFmappingLoad • 1–135
- PMDFoptionDispose • 1–137
- PMDFoptionGetInteger • 1–138
- PMDFoptionGetReal • 1–140
- PMDFoptionGetString • 1–142
- PMDFoptionRead • 1–144
- PMDFqueueCacheEnd • 1–146
- PMDFqueueCacheGetEntry • 1–147
- PMDFreadFailureLog • 1–151
- PMDFreadHeader • 1–153
- PMDFreadLine • 1–154
- PMDFreadText • 1–156
- PMDFreceiptControl • 1–158
- PMDFrecipientDisposition • 1–161
- PMDFreturnMessage • 1–164
- PMDFrewindMessage • 1–167
- PMDFsetCallBack • 1–168
- PMDFsetEnvelopeId • 1–170
- PMDFsetLimits • 1–172
- PMDFsetMutex • 1–174
- PMDFsetReceiptAddresses • 1–181
- PMDFsetRecipientFlags • 1–177
- PMDFsetRecipientType • 1–179
- PMDFstartMessageBody • 1–183
- PMDFstartMessageEnvelope • 1–184

Routines (cont'd)

- PMDFstartMessageHeader • 1–186
- PMDFwriteDate • 1–187
- PMDFwriteFrom • 1–188
- PMDFwriteHeader • 1–190
- PMDFwriteLine • 1–191
- PMDFwriteSubject • 1–193
- PMDFwriteText • 1–195
- PMDF_abort_message • 1–4
- PMDF_add_recipient • 1–3
- PMDF_close_log_file • 1–10
- PMDF_close_queue_cache • 1–10
- PMDF_dequeue_end • 1–5
- PMDF_dequeue_initialize • 1–5
- PMDF_dequeue_message_end • 1–5
- PMDF_enqueue_initialize • 1–3
- PMDF_enqueue_message • 1–3
- PMDF_get_channel_name • 1–4
- PMDF_get_envelope_id • 1–4
- PMDF_get_message • 1–5
- PMDF_get_recipient • 1–5
- PMDF_get_recipient_flags • 1–4, 1–5
- PMDF_log • 1–12
- PMDF_receipt_control • 1–181
- PMDF_recipient_disposition • 1–5
- PMDF_set_call_back • 1–10
- PMDF_set_envelope_id • 1–4
- PMDF_set_recipient_flags • 1–4
- PMDF_start_message_envelope • 1–3
- PMDF_start_message_header • 1–3
- summary of API routines • 1–36

S

Sending mail

- See enqueuing messages
- Sensitivity: header line • 2–38
- SHORTALFA_SIZE = 40 bytes • 1–41
- Size limits • 1–172
- slave_debug keyword • 1–103
- SS\$_NOWORLD • A–5

Stopping

- dequeue • 1–82, 1–85
- enqueue • 1–42

Strings • 1–36, 1–40

- Subject: header line • 1–193, 2–38

Submitting mail

- See enqueuing messages

Summary

- API routines • 1–36
- PMDF_send item codes • 2–17

T

Threads • 1–7, 1–174
Time • 1–105
To: addresses
 NOTARY flags • 1–122, 1–177
 reading • 1–119
 specifying • 1–47, 2–38
 writing • 1–47

U

Unique string, obtaining • 1–124
User name, obtaining • 1–126

W

Warnings-to: header line • 2–39
Writing message lines • 1–191, 1–195

X

X-Organization: header line • 2–39
X-PS-qualifiers: header line • 2–39

