

DIGITAL UNIX

Asynchronous Transfer Mode

Part Number: AA-QDP5D-TE

December 1997

Product Version: DIGITAL UNIX Version 4.0D or higher

This manual is for experienced UNIX kernel programmers responsible for writing Asynchronous Transfer Mode (ATM) device drivers and kernel modules. It describes the DIGITAL UNIX ATM subsystem, how to configure the subsystem, and how to use the ATM kernel interfaces.

© Digital Equipment Corporation 1997
All rights reserved.

The following are trademarks of Digital Equipment Corporation: ALL-IN-1, Alpha AXP, AlphaGeneration, AlphaServer, AltaVista, ATMworks, AXP, Bookreader, CDA, DDIS, DEC, DEC Ada, DEC Fortran, DEC FUSE, DECnet, DECstation, DECsystem, DECterm, DECUS, DECwindows, DTIF, Massbus, MicroVAX, OpenVMS, POLYCENTER, Q-bus, StorageWorks, TruCluster, ULTRIX, ULTRIX Mail Connection, ULTRIX Worksystem Software, UNIBUS, VAX, VAXstation, VMS, XUI, and the DIGITAL logo.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

All other trademarks and registered trademarks are the property of their respective holders.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii).

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from DIGITAL or an authorized sublicensor.

Digital conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

Contents

About This Manual

1 Overview of DIGITAL UNIX ATM Architecture

1.1	ATM Subsystem	1-1
1.1.1	Connection Management Module	1-4
1.1.2	CMM Interfaces	1-4
1.2	ATM Subsystem Configuration	1-5

2 Configuring the ATM Software

2.1	ATM Environment	2-1
2.1.1	Classical IP Environment	2-1
2.1.2	LAN Emulation Environment	2-2
2.1.3	IP Switching	2-3
2.2	ATM Planning	2-5
2.2.1	Verifying That the ATM Subsets Are Installed	2-5
2.2.2	Configuring ATM into the Kernel	2-5
2.2.3	Preparing for the Configuration	2-6
2.2.3.1	Classical IP Information	2-6
2.2.3.2	LAN Emulation Information	2-9
2.2.3.3	IP Switching Information	2-11
2.3	Configuring ATM	2-12
2.3.1	Configuring Classical IP	2-13
2.3.1.1	Editing the /etc/atmhosts File	2-13
2.3.1.2	Adding Hosts to the hosts Database	2-14
2.3.1.3	Running atmsetup	2-15
2.3.1.4	Running netsetup (SVCs only)	2-16
2.3.1.5	Editing the /etc/atm.conf File (PVCs only)	2-16
2.3.1.6	Running atmconfig (PVCs only)	2-18
2.3.2	Configuring LAN Emulation	2-19
2.3.2.1	Editing the /etc/atmhosts File	2-19
2.3.2.2	Adding Hosts to the hosts Database	2-20
2.3.2.3	Running atmsetup	2-21
2.3.2.4	Running netsetup	2-22
2.3.3	Configuring IP Switching	2-22

2.3.3.1	Editing the /etc/hosts File	2-23
2.3.3.2	Running atmsetup	2-23
2.3.3.3	Editing the /etc/atm.conf File	2-24
2.3.3.4	Running atmconfig	2-26
2.4	Managing the ATM Environment	2-26
2.4.1	Managing ATM Networking and Displaying Information About ATM Networks	2-26
2.4.2	Managing the Signaling Module	2-27
2.4.3	Managing the Classical IP Environment	2-27
2.4.4	Managing the LAN Emulation Environment	2-28
2.4.4.1	Managing LAN Emulation Clients	2-28
2.4.4.2	Displaying the LE-ARP Table	2-28
2.4.5	Managing IP Switching	2-28
2.5	Solving Problems	2-28
2.5.1	Using the Diagnostic Map	2-29
2.5.2	Getting Started	2-29
2.5.3	Solving ATM Problems	2-30
2.5.3.1	Solving CLIP Problems	2-31
2.5.3.2	Solving LANE Problems	2-35
2.5.3.3	Solving IP Switching Problems	2-38

3 ATM Subsystem General Features

3.1	Header Files	3-1
3.2	Kernel Data File	3-2
3.3	ATM Module Configuration	3-2
3.4	Error Codes	3-2
3.5	Data Formats	3-3
3.5.1	Raw ATM Cells	3-3
3.5.2	Cooked Data	3-4
3.5.3	How the Data Is Carried	3-4
3.5.4	Time-Stamping	3-4
3.6	Physical Point of Attachment	3-5
3.6.1	Permanent Virtual Circuit PPAs	3-7
3.6.2	Switched Virtual Circuit PPAs	3-8
3.7	Memory Allocation	3-9
3.8	ATM Locking Macros	3-10
3.8.1	Locking Guidelines	3-10
3.8.2	Types of Locking Methods	3-11
3.8.3	Order of Locking Macros	3-12
3.8.4	Creation of ATM Threads	3-13
3.9	Types of Circuits	3-14

3.10	Global Data Structures	3-14
3.10.1	The atm_vc Structure	3-15
3.10.1.1	The conv_pp1 and conv_pp2 Members	3-17
3.10.1.2	The sig_pp1, sig_pp2, drv_pp1, and drv_pp2 Members	3-17
3.10.1.3	The vcs Member	3-17
3.10.1.4	The call_reference Member	3-17
3.10.1.5	The errno Member	3-17
3.10.1.6	The vci and vpi Members	3-18
3.10.1.7	The ppa Member	3-18
3.10.1.8	The selector Member	3-18
3.10.1.9	The direction Member	3-18
3.10.2	The atm_addr Structure	3-18
3.10.2.1	The vc Member	3-20
3.10.2.2	The address Member	3-20
3.10.2.3	The ton Member	3-20
3.10.2.4	The anpi Member	3-20
3.10.2.5	The subaddress Member	3-20
3.10.2.6	The subaddress_type Member	3-20
3.10.2.7	The eprtype Member	3-20
3.10.2.8	The endpoint Member	3-21
3.10.2.9	The state Member	3-21
3.10.2.10	The atm_error Member	3-21
3.10.2.11	The setup Member	3-21
3.10.2.12	The connect Member	3-21
3.10.2.13	The location, cause, diag_length, and diagnostic Members	3-22
3.10.2.14	The endstate Member	3-22
3.10.2.15	The conv_p1 and conv_p2 Members	3-22
3.10.2.16	The sig_p1 and sig_p2 Members	3-22
3.10.2.17	Allocating the atm_addr Structure	3-22
3.10.3	The atm_vc_services Structure	3-23
3.10.3.1	The vc Member	3-25
3.10.3.2	The fqos and bqos Members	3-25
3.10.3.3	The fntu and bmtu Members	3-25
3.10.3.4	The valid_rates Member	3-26
3.10.3.5	The fpeakcr, bpeakcr, fsustcr, bsustcr, fburstcr, and bburstcr Members	3-26
3.10.3.6	The flags Member	3-27
3.10.3.7	The aal Member	3-28
3.10.3.8	The queue Member	3-28

3.10.3.9	The bearer_class Member	3-28
3.10.3.10	The lerrstat Member	3-28
3.10.3.11	The nerrstat Member	3-30
3.10.3.12	The cmm_drv_handle Member	3-30
3.10.3.13	The drv_resource Member	3-30
3.10.3.14	The converge_handle Member	3-31
3.10.3.15	Allocating the atm_vc_services Structure	3-31
3.10.4	The atm_uni_call_ie Structure	3-31
3.10.4.1	The ie_type Member	3-33
3.10.4.2	The last Member	3-33
3.10.4.3	The aal_params Member	3-34
3.10.4.4	The bb_high_layer and bb_low_layer Members	3-34
3.10.4.5	Allocating the atm_uni_call_ie Structure	3-34
3.10.4.6	Setting Fields in the atm_uni_call_ie Structure	3-36
3.10.5	The atm_ppa Structure	3-37
3.10.5.1	The driver Member	3-38
3.10.5.2	The sig Member	3-38
3.10.5.3	The ppas_id Member	3-38
3.10.5.4	The ton Member	3-38
3.10.5.5	The anpi Member	3-38
3.10.5.6	The addrlen Member	3-38
3.10.5.7	The address Member	3-39
3.10.5.8	The uni Member	3-39
3.10.5.9	The type Member	3-39
3.10.5.10	The esi_arg Member	3-39
3.10.6	The atm_esi Structure	3-39
3.10.6.1	The esi and esilen Members	3-40
3.10.6.2	The driver Member	3-40
3.10.6.3	The sigp1 and sigp2 Members	3-40
3.10.7	The atm_cause_info Structure	3-40
3.10.7.1	The cause Member	3-41
3.10.7.2	The location Member	3-42
3.10.7.3	The module_name Member	3-42
3.10.7.4	The reason Member	3-42
3.10.7.5	The diag_length Member	3-42
3.10.7.6	The diag Member	3-42

4 Device Driver Interface

4.1	Registering the Device Driver	4-1
4.2	Receiving Data Packets and Cells	4-2
4.3	Reporting Errors	4-2

4.4	Using ATM Device Driver Interface Structures	4-2
4.4.1	The atm_drv_params Structure	4-3
4.4.1.1	The name member	4-4
4.4.1.2	The unit Member	4-4
4.4.1.3	The type Member	4-4
4.4.1.4	The num_vc Member	4-4
4.4.1.5	The max_vcib and max_vpib Members	4-4
4.4.1.6	The max_vci and max_vpi Members	4-4
4.4.1.7	The sent Member	4-4
4.4.1.8	The received Member	4-5
4.4.1.9	The dropped Member	4-5
4.4.1.10	The num_vci and num_vpi Members	4-5
4.4.1.11	The hard_mtu Member	4-5
4.4.1.12	The nqueue Member	4-5
4.4.1.13	The flowcontrol Member	4-5
4.4.1.14	The rates Member	4-5
4.4.1.15	The capabilities Member	4-6
4.4.1.16	The numid Member	4-7
4.4.1.17	The ids Member	4-7
4.4.2	The atm_queue_param Structure	4-7
4.4.2.1	The vc Member	4-8
4.4.2.2	The qlength Member	4-8
4.4.2.3	The qtime Member	4-8
4.4.2.4	The flags Member	4-8

5 Signaling Module Interface

5.1	Registering the Signaling Module	5-2
5.2	Receiving a New Call	5-2
5.3	Reporting a VC Activation	5-2
5.4	Activating a Connection	5-2
5.5	Reporting a Connection Failure	5-3
5.6	Releasing a Connection	5-3
5.7	Dropping an Endpoint	5-3
5.8	Deleting a Connection	5-3
5.9	Restarting a Virtual Circuit	5-3
5.10	Reporting a Completed Restart	5-4
5.11	Reporting a Completed Status Enquiry	5-4
5.12	Requesting Endpoint Information	5-4
5.13	Adding a PPA	5-4
5.14	Deleting a PPA	5-5
5.15	Reporting Completed MIB Access	5-5

5.16	Requesting VC Status	5-5
5.17	Using the atm_oid Structure	5-5

6 Convergence Module Interface

6.1	Registering a Convergence Module	6-1
6.2	Receiving Data	6-2
6.2.1	Receiving Exception Notifications	6-2
6.2.2	Connecting to the ATM Module Management Interface ..	6-2
6.3	Unregistering a Convergence Module	6-3
6.4	Requesting Interface Parameters	6-3
6.5	Reserving Resources for CBR and Pacing Circuits	6-4
6.6	Releasing Reserved Resources	6-4
6.7	Requesting a Connection to a Remote System	6-4
6.8	Adding an Endpoint to a Connection	6-5
6.9	Requesting a Connection Be Torn Down	6-5
6.10	Dropping an Endpoint from a Connection	6-6
6.11	Transmitting Data on an Established VC	6-6
6.12	Modifying VC Parameters	6-6
6.13	Requesting Endpoint Connection State Information	6-6
6.14	Binding to a PPA	6-6
6.15	Receiving a Connection Notification	6-7
6.16	Unbinding from a PPA	6-8
6.17	Accepting an Incoming Call	6-8
6.18	Rejecting an Incoming Call	6-8
6.19	Adding a New ATM Address	6-8
6.20	Deleting an ATM Address	6-9
6.21	Requesting VC Statistics	6-9
6.22	Using ATM Convergence Module Interface Structures	6-9
6.22.1	The atm_vc_stats Structure	6-9
6.22.2	The atm_cmi_addr Union	6-10
6.22.2.1	The addr Member	6-11
6.22.2.2	The vcn Member	6-11

7 Connections

7.1	Making Outgoing Connections	7-1
7.1.1	Making the Call	7-2
7.1.2	Adding Parties to an Existing Connection	7-4
7.2	Accepting Connections	7-5
7.3	Controlling the Aging of Connections	7-6
7.4	Releasing a Connection	7-8

7.4.1	Release by a Convergence Module	7-8
7.4.2	Release by Network or Endpoint	7-9
7.5	Creating Permanent Virtual Circuits	7-9
7.6	Creating Signaling Virtual Circuits	7-10
8	Module Management Interface	
8.1	Creating an MMI Path	8-2
8.2	Defining New MMI ioctl Commands	8-3
8.3	Using MMI Calling Conventions	8-5
8.4	Using the Device Driver MMI	8-5
8.5	Using the Signaling Module MMI	8-6
8.6	Using the Convergence Module MMI	8-6
9	Queuing Guidelines	
9.1	Queuing in Device Drivers	9-1
9.1.1	Device Driver Transmit Queuing	9-1
9.1.2	Device Driver Receive Queuing	9-3
9.2	Queuing in Convergence Modules	9-3
9.2.1	Convergence Module Transmit Queuing	9-3
9.2.2	Convergence Module Receive Queuing	9-4
10	Flow Control	
10.1	Hardware Flow Control	10-1
10.2	Software Flow Control	10-2
10.2.1	High-Water Mark	10-2
10.2.2	Low-Water Mark	10-3
10.3	Convergence Module Flow Control	10-3
11	Tuning the ATM Subsystem	
A	CMM Routines	
	atm_cmm_accept	A-2
	atm_cmm_activate_con	A-4
	atm_cmm_add	A-6
	atm_cmm_adi_set_cause	A-8
	atm_cmm_adi_set_log	A-10
	atm_cmm_alloc_addr	A-12
	atm_cmm_alloc_ie	A-13
	atm_cmm_alloc_services	A-15
	atm_cmm_bind_info	A-16

atm_cmm_con_deleted	A-21
atm_cmm_con_failed	A-23
atm_cmm_con_release	A-25
atm_cmm_connect	A-27
atm_cmm_del_esi	A-31
atm_cmm_del_ppa	A-32
atm_cmm_drop	A-34
atm_cmm_enquery	A-35
atm_cmm_ep_dropped	A-36
atm_cmm_error	A-38
atm_cmm_findaddr	A-40
atm_cmm_find_driver	A-42
atm_cmm_free_addr	A-43
atm_cmm_free_ie	A-44
atm_cmm_free_services	A-45
atm_cmm_mib_response	A-47
atm_cmm_new_call	A-49
atm_cmm_new_esi	A-52
atm_cmm_new_ppa	A-54
atm_cmm_new_thread	A-57
atm_cmm_next_cause	A-59
atm_cmm_oam_receive	A-61
atm_cmm_ppa_bind	A-62
atm_cmm_ppa_info	A-65
atm_cmm_ppa_unbind	A-69
atm_cmm_receive	A-70
atm_cmm_register_cvg	A-73
atm_cmm_register_dd	A-76
atm_cmm_register_sig	A-78
atm_cmm_reject	A-82
atm_cmm_release	A-83
atm_cmm_reply	A-85
atm_cmm_reserve_resources	A-87
atm_cmm_restart	A-89
atm_cmm_restart_ack	A-91
atm_cmm_send	A-93
atm_cmm_set_cause	A-95
atm_cmm_set_log	A-97
atm_cmm_smi_set_cause	A-99
atm_cmm_smi_set_log	A-101
atm_cmm_status_done	A-103
atm_cmm_unregister_cvg	A-104
atm_cmm_vc_control	A-106

atm_cmm_vc_get	A-108
atm_cmm_vc_stats	A-109
xxx_add	A-110
xxx_connect	A-111
xxx_drop	A-116
xxx_enquery	A-117
xxx_except	A-118
xxx_manage	A-124
xxx_mib	A-127
xxx_mmi	A-129
xxx_receive	A-131
xxx_release	A-134
xxx_restart	A-135
xxx_setup	A-137
xxx_xmit	A-139

B Connection Programming Examples

B.1 Making a Call	B-1
B.2 Adding More Parties to a Point-to-Multipoint Connection	B-3
B.3 Processing an Incoming Call	B-5

Index

Examples

3-1 The atm_uni_call_ie Structure Definition	3-32
B-1 Making a Call Code Fragment	B-1
B-2 Adding Parties to a Point-to-Multipoint Connection Code Fragment	B-3
B-3 Incoming Call Processing Code Fragment	B-5

Figures

1-1 DIGITAL UNIX ATM Subsystem	1-3
2-1 Classical IP over an ATM Network	2-2
2-2 Emulated LAN over an ATM Network	2-3
2-3 IP Switching over an ATM Network	2-4
2-4 Configuration Worksheet, Part A	2-7
2-5 Configuration Worksheet, Part B	2-9

2-6	Configuration Worksheet, Part C	2-11
-----	---------------------------------------	------

Tables

2-1	ATM Kernel Options	2-5
3-1	The atm_vc Structure Members	3-16
3-2	The atm_addr Structure Members	3-19
3-3	The atm_vc_services Structure Members	3-24
3-4	Information Element Macros	3-36
3-5	The atm_ppa Structure Members	3-37
3-6	The atm_esi Structure Members	3-40
3-7	The atm_cause_info Structure Members	3-41
4-1	The atm_drv_params Structure Members	4-3
4-2	The atm_queue_param Structure Members	4-8
5-1	The atm_oid Structure Members	5-6
6-1	The atm_vc_stats Structure Members	6-10
7-1	Aging Parameter Values	7-7
8-1	The atm_mmi_path Structure Members	8-3
11-1	ATM Subsystem Configuration Attributes	11-1
11-2	UNI3X Subsystem Configuration Attributes	11-2
11-3	LANE Subsystem Configuration Attributes	11-2

About This Manual

This manual describes the DIGITAL UNIX[®] Asynchronous Transfer Mode (ATM) subsystem, how to configure the subsystem, and how to use the ATM kernel interfaces. This document does not describe the application programming interface (API) that user-level applications would use to access the ATM subsystem. Also, this manual is not an ATM networking tutorial.

After reading this manual, you should be able to:

- Understand the ATM subsystem architecture
- Configure the ATM software
- Understand how the different kernel interfaces operate
- Write a kernel module

Audience

This manual is for experienced UNIX kernel programmers responsible for writing device drivers and kernel modules. These programmers should be familiar with the following:

- ATM technology
- ATM Forum User-Network Interface (UNI) Version 3.0 specification
- C language programming

The secondary audience is system administrators responsible for configuring network software. These system administrators should be familiar with the following:

- ATM technology
- C language
- Programming interfaces for UNIX operating systems

New and Changed Features

This manual has been revised to document all of the Asynchronous Transfer Mode changes that are part of the current release. This includes

new information on the types of ATM configurations and step-by-step configuration instructions for setting up Classical IP, LAN Emulation, and IP switching environments.

Organization

This manual is organized into eleven chapters and two appendixes.

Chapter 1	Provides an overview of the DIGITAL UNIX Asynchronous Transfer Mode (ATM) architecture and its kernel interfaces.
Chapter 2	Describes how to configure the ATM software on a DIGITAL UNIX system.
Chapter 3	Describes the ATM header files, generic data structures, macros, and return codes that ATM modules use.
Chapter 4	Describes the ATM device driver interface, its tasks and routines, and associated data structures.
Chapter 5	Describes the ATM signaling module interface, its tasks and routines, and associated data structures.
Chapter 6	Describes the ATM convergence module interface, its tasks and routines, and associated data structures.
Chapter 7	Describes how ATM connections are initiated and terminated, and includes some code fragments that show how these tasks are implemented in software.
Chapter 8	Describes the ATM Module Management Interface (MMI).
Chapter 9	Describes queuing information that kernel module writers require.
Chapter 10	Describes the flow control in the ATM subsystem.
Chapter 11	Describes how to tune the ATM subsystem.
Appendix A	Describes the ATM CMM routines in reference-page format.
Appendix B	Contains programming code fragments that show certain connection-related tasks.

Related Documents

For information about DIGITAL UNIX device driver programming, refer to the following manuals that are part of the Device Driver Documentation kit:

- *Writing Device Drivers: Tutorial*
- *Writing Device Drivers: Advanced Topics*
- *Writing Device Drivers: Reference*
- *Writing Network Device Drivers*

- *Writing TURBOchannel Device Drivers*
- *Writing EISA and ISA Bus Device Drivers*
- *Writing VMEbus Device Drivers*
- *Writing PCI Bus Device Drivers*
- *Writing Device Drivers for the SCSI/CAM Architecture Interfaces*

For additional information about ATM, refer to the *ATM User-Network Interface Specification, Version 3.0* ISBN 0-13-225863-3 and the *ATM User-Network Interface Specification, Version 3.1* ISBN 0-13-393828-X, both published by Prentice-Hall.

For information on installing the DGLTA ATM adapter and its device driver, see the *ATMworks 350 Adapter Installation and Service* guide.

For information about administering networking interfaces, refer to the *System Administration* guide and the *Network Administration* guide.

The printed version of the DIGITAL UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from DIGITAL.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General users	G	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the DIGITAL UNIX documentation set.

Reader's Comments

DIGITAL welcomes any comments and suggestions you have on this and other DIGITAL UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

A Reader's Comment form is located on your system in the following location:

`/usr/doc/readers_comment.txt`

- Mail:

Digital Equipment Corporation
 UBPG Publications Manager
 ZKO3-3/Y32
 110 Spit Brook Road
 Nashua, NH 03062-9987

A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of DIGITAL UNIX that you are using.
- If known, the type of processor that is running the DIGITAL UNIX software.

The DIGITAL UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate DIGITAL technical support office. Information provided with the software media explains how to send problem reports to DIGITAL.

Conventions

This document uses the following typographic conventions:

- | | |
|--------------|---|
| % | |
| \$ | A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells. |
| # | A number sign represents the superuser prompt. |
| % cat | Boldface type in interactive examples indicates typed user input. |

file

Italic (slanted) type indicates variable values, placeholders, and function argument names.

[|]
{ | }

In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

...

In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.

cat(1)

A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat(1)` indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Return

In an example, a key name enclosed in a box indicates that you press that key.

Ctrl/x

This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).

Overview of DIGITAL UNIX ATM Architecture

Asynchronous Transfer Mode (ATM) networks promise to become the dominant network interconnect because they provide the following capabilities:

- Speeds from 25 M/bps to up to 622 M/bps or greater through cell-switching.
- Multiple qualities of service.
- Connection-oriented interconnection with resource reservation for individual connections. These connections might be for conversations between two applications or for a connection over which many conversations between many applications and protocols are multiplexed.

Presently, interest in implementing ATM networks, particularly in the local area, comes from applications that need the high speed and the low latency (switched, full duplex network infrastructure) that ATM networks provide.

Caution

Because of changes in the DIGITAL UNIX kernel, ATM code compiled under previous versions of the operating system must be recompiled. In addition, because of changes to the ATM interfaces, you should verify and change, if necessary, the correct number and type of arguments in ATM function calls before recompiling.

1.1 ATM Subsystem

The DIGITAL UNIX ATM subsystem is a separately configurable kernel subsystem with the following characteristics:

- Provides a set of ATM-related services to kernel and user applications. These applications, in turn, must provide their own interface to ATM.
- Provides a well-defined set of interfaces for using these services. Chapter 3 through Chapter 10 describe the interfaces and services.

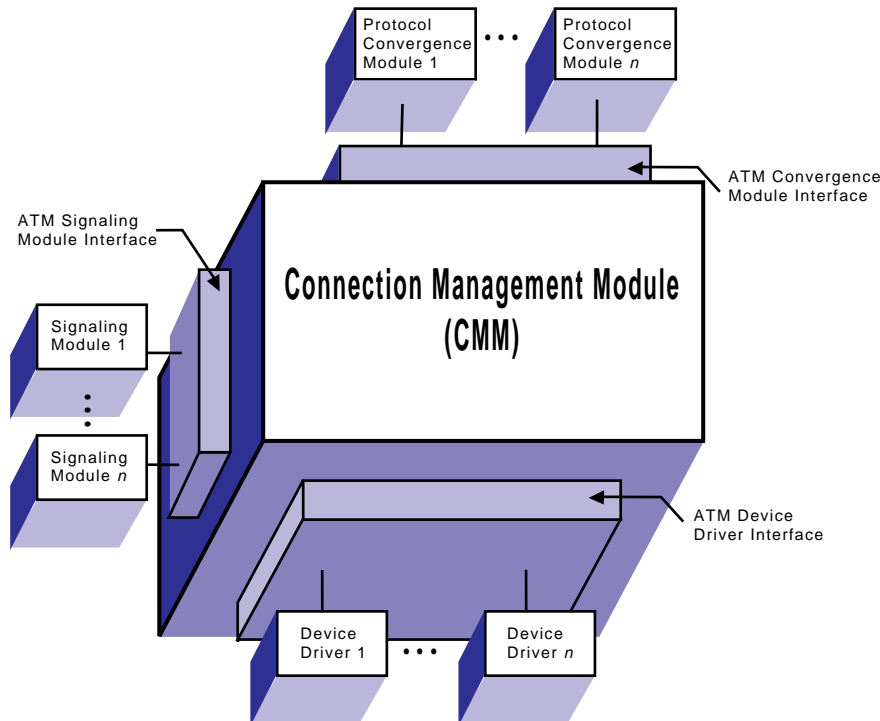
- Is optimized for ATM functions; does not provide support for a specific set of user protocols.
- Is flexible and easy to expand and adapt to changing requirements.
- Provides an open interface for ATM hardware designers and for software developers to interface new or existing protocols to ATM.

The DIGITAL UNIX ATM subsystem consists of the following parts:

- **Connection Management Module (CMM)**
This module handles all communications between the various elements of the ATM subsystem as well as managing all virtual circuits (VCs) and communications with protocol stacks that use ATM.
- **One or more ATM hardware adapter device drivers**
These device drivers handle the hardware-specific details for controlling specific adapters.
- **One or more signaling protocol modules**
These modules implement specific protocols for communicating connection management information between the end system and the switch. In this implementation, UNI 3.0/3.1 is the default signaling protocol module that the DIGITAL UNIX ATM subsystem uses.

Figure 1-1 illustrates the ATM subsystem.

Figure 1–1: DIGITAL UNIX ATM Subsystem



ZK-1022U-AI

The DIGITAL UNIX ATM subsystem supports any number of ATM device drivers and signaling protocol modules as long as the actions of these modules do not interfere with each other. ATM modules should communicate only with the CMM as the CMM coordinates all communications between ATM modules. ATM modules should never bypass the CMM and communicate with each other directly.

The three elements of the ATM subsystem provide only raw ATM services. They do not implement any specific protocol for carrying data over the ATM network. That is, the ATM subsystem is entirely protocol independent. If a specific protocol wants to use the ATM network to transport data, the protocol must provide an interface into the ATM subsystem.

The programs that provide the interface between the ATM subsystem and specific protocols are called convergence modules. These modules, using the ATM subsystem interface Kernel Programming Interface (KPI), control the creation and use of ATM virtual circuits (VCs) in a manner appropriate for the protocol. Since different protocols have different requirements, the ATM subsystem leaves the adaptation of the protocol to use ATM entirely up to

the protocol implementor. The ATM subsystem does not provide any protocol-specific services.

1.1.1 Connection Management Module

The CMM is the central module in the DIGITAL UNIX ATM subsystem and has the following duties:

- Manages all activities of the ATM subsystem, including subsystem configuration. All data to and from the ATM networks must pass through the CMM to get properly routed to the correct destination.
- Handles all circuit connection requests from protocol modules (convergence modules) and manages all VCs from the time they are created until they are torn down.
- Provides interface points for other elements of the ATM subsystem, as well as the interfaces for accessing the ATM subsystem from kernel protocol stacks. These interfaces permit the modular addition of drivers, signaling modules, management modules, and protocol stack interfaces without having to make changes to the CMM.
- Provides the only interface into the ATM subsystem from the DIGITAL UNIX kernel. ATM modules are accessed only through the CMM.

1.1.2 CMM Interfaces

The CMM provides the following interfaces:

- ATM device driver interface
ATM adapter device drivers use this interface to access the ATM subsystem. See Chapter 4 for more information.
- Signaling module interface
Modules that implement ATM signaling protocols use this interface. This interface allows multiple signaling modules to provide services to protocol stacks on a per-VC basis, as long as the signaling modules do not conflict on the ATM network. See Chapter 5 for more information.
- Convergence module interface
Kernel protocol stacks use this external interface to the ATM subsystem to gain access to the ATM subsystem services. See Chapter 6 for more information.

These interfaces are registration-type interfaces. Each makes calls to the CMM to inform the CMM of its presence. The CMM places no limit on the number of each type of module that may register, the total number being limited only by system resources.

1.2 ATM Subsystem Configuration

Before you communicate on the ATM network, you must install the DIGITAL DGLTA ATM adapter and configure the ATM software. See the installation and service documentation shipped with the adapter for information on installing the ATM adapter and Chapter 2 for information on configuring the ATM software.

Configuring the ATM Software

After you install the DIGITAL DGLTA adapter and DIGITAL UNIX ATM subsets, you must configure the software in order to communicate on the ATM network to which the system is connected. This chapter describes the DIGITAL UNIX ATM network environment and how to configure the ATM subsystem. In addition, this chapter describes the commands for managing the ATM subsystem and contains a diagnostic map to help you isolate any problems that might occur.

2.1 ATM Environment

An ATM network consists of the following:

- Switch

A specialized system that maintains a list of virtual channel identifiers (VCIs) and virtual path identifiers (VPIs), connects one end system to another, and forwards or switches ATM cells from one end system to another based on the VCI/VPI information contained in the cell

- End system

A system physically connected to a switch that communicates with other end systems through the switch

In the DIGITAL UNIX ATM environment, the following configurations are possible:

- Classical Internet Protocol (CLIP)
- Local Area Network (LAN) emulation
- IP switching

The following sections describe each of these configurations and the roles of systems in each.

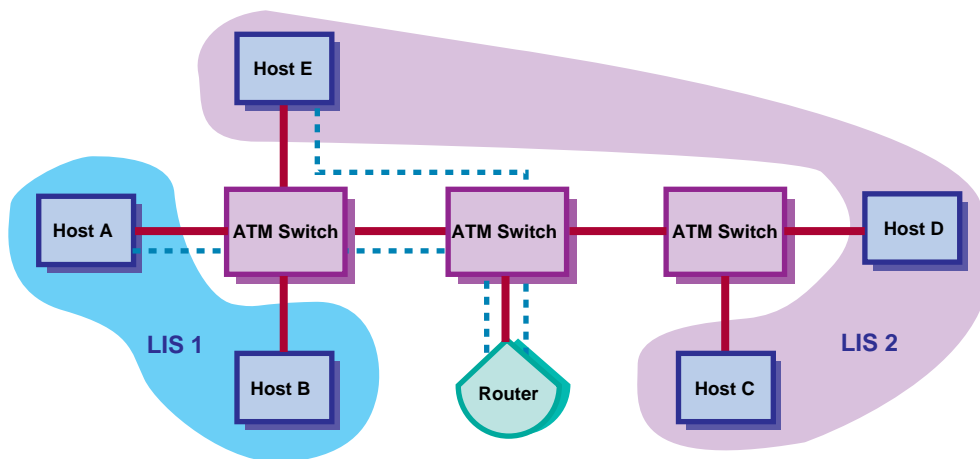
2.1.1 Classical IP Environment

The Classical IP environment, as described in RFC 1577, provides a basic means for carrying unicast IP traffic over ATM networks. In this environment, hosts that can communicate with each other are grouped into a Logical IP Subnetwork (LIS). An ATM network can contain multiple LISs. In a LIS, all hosts and routers have the following requirements:

- Have the same IP network/subnetwork number and mask.
- Are directly connected to the ATM network.
- Access members outside the LIS through a router.
- For switched virtual circuits (SVCs), use Address Resolution Protocol (ARP) to resolve IP protocol addresses to ATM hardware addresses. For SVCs and permanent virtual circuits (PVCs), use Inverse ARP to resolve ATM hardware addresses to IP protocol addresses.
- Can communicate with all other members in the same LIS (mesh topology).

Figure 2–1 shows an ATM network with two LISs. Host A and Host B are members of LIS 1; Host C, Host D, and Host E are members of LIS 2. The figure also shows a virtual circuit (VC) between Host A and the router and between Host E and the router. Although these hosts are connected to the same switch and might establish a VC for communications between each other, they cannot because all communications to a member of another LIS must go through a router.

Figure 2–1: Classical IP over an ATM Network



ZK-1307U-AI

2.1.2 LAN Emulation Environment

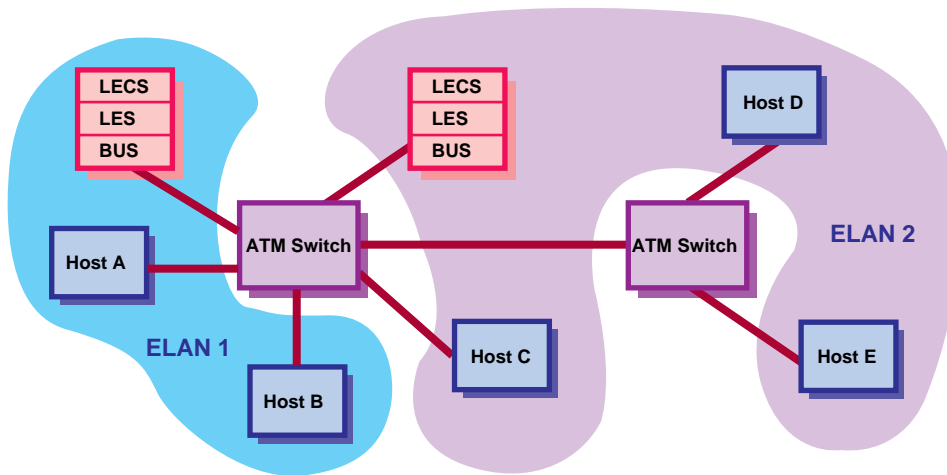
The LAN Emulation (LANE) environment, as defined by the ATM Forum, groups hosts into an entity called an emulated LAN. A LANE environment has the following characteristics:

- Identifies hosts through their 48-bit media access control (MAC) addresses

- Supports multicast and broadcast services either through point-to-multipoint connections or through a multicast server, unlike the Classical IP environment
- Supports any protocol that uses an IEEE broadcast LAN

Figure 2–2 shows an ATM network with two emulated LANs. Host A and Host B are LAN Emulation Clients (LECs) on ELAN1. Host C, Host D, and Host E are LECs on ELAN2. The LECS (LAN Emulation Configuration Server), the LES (LAN Emulation Server), and the BUS (Broadcast and Utility Server) are depicted as two separate systems, although these server functions are typically resident on an ATM switch.

Figure 2–2: Emulated LAN over an ATM Network



ZK-1323U-AI

2.1.3 IP Switching

The IP switching environment consists of one or more hosts connected to an IP switch. Each host is connected to the IP switch through a point-to-point physical connection, with each physical connection as a separate subnet. Communication between the host and the IP switch occurs over dynamically created PVCs.

The IP switch is a typical ATM switch with added IP controller software that performs IP routing and IP traffic classification functions. In this environment, a series of packets from one host to another with the same protocol type, type of service, and other characteristics indicated in the packet header is called a **flow**. When the IP controller identifies a flow that is of long duration, it instructs the ATM switch to make the appropriate hardware connections and to forward the ATM cells directly to the

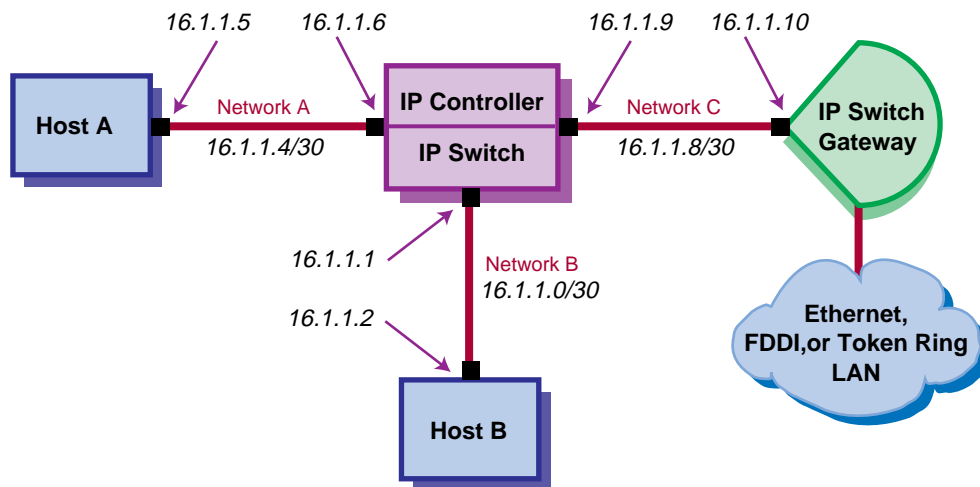
destination, bypassing the IP controller. This increases throughput at the switch and throughout the network.

The DIGITAL UNIX IP switching implementation is based on the Ipsilon Networks, Inc. reference model and has the following characteristics:

- Supports IP traffic only
- Supports multicast and broadcast services
- Does not require systems to function as ARP servers or multicast servers
- Uses the Ipsilon Flow Management Protocol (IFMP) to exchange control information with the IP switch
- Does not require that ATM Forum signaling (options UNI3X) be configured on the system
- Requires fewer configuration steps than Classical IP and LAN emulation

Figure 2-3 shows a simple ATM network with an IP switch, IP switch gateway, some hosts, and a legacy LAN network. Host A, Host B, and the IP gateway are on separate subnets. The IP switch gateway runs a routing protocol and advertises routes to other subnets to hosts on the legacy LAN.

Figure 2-3: IP Switching over an ATM Network



ZK-1305U-AI

For the IP switching subnetworks, DIGITAL recommends that you use a network mask length of 30 bits. This allows for two bits for each host address, one bit for the subnetwork address, and one bit for the broadcast

address. Using large netmasks helps to conserve IP address space on subnetworks that have a few hosts attached.

2.2 ATM Planning

This section describes those tasks that you need to do before configuring the ATM software.

2.2.1 Verifying That the ATM Subsets Are Installed

Verify that the ATM subsets are installed by entering the following command:

```
# setld -i | grep ATM
```

If all of the subsets are not installed, install them by using the `setld` command. For more information on installing subsets, see `setld(8)`, the *Installation Guide*, or the *System Administration* manual.

Note

You do not have to install the `OSFATMBINOBJECT` subset.

2.2.2 Configuring ATM into the Kernel

After you install the ATM subsets, verify that the ATM support you require is in the kernel by issuing the following command:

```
# sysconfig -q atm
```

If `atm:` is not displayed, log in as superuser and complete the following steps:

1. Build a new kernel by issuing the `doconfig` command. If you are unfamiliar with rebuilding the kernel, see the *System Administration* manual.
2. When prompted, select one or more of the kernel options in Table 2–1:

Table 2–1: ATM Kernel Options

Option	Purpose
<code>options ATM</code>	For base ATM support (required)
<code>options UNI3X</code>	For ATM Forum signaling with either LANE or Classical IP
<code>options ATMILMI3X</code>	For ATM Forum Integrated Layer Management Interface (ILMI) support

Table 2–1: ATM Kernel Options (cont.)

<code>options ATMIP</code>	For Classical IP services
<code>options LANE</code>	For ATM Forum LAN Emulation (LANE)
<code>options ATMIFMP</code>	For IP switching

Note

If the ATM hardware is already installed, `options ATM` is automatically selected as a mandatory option.

3. Reboot your system with the new kernel by issuing the following command:

```
# shutdown -r now
```

This command immediately shuts down and automatically reboots the system.

2.2.3 Preparing for the Configuration

After you verify ATM support in the kernel, you configure ATM. ATM configuration can consist of the following parts:

- Configuring for Classical IP
- Configuring for LAN Emulation
- Configuring for IP switching

The type of information you need depends on the types of environment you want to set up and use.

2.2.3.1 Classical IP Information

Figure 2–4 shows Part A of the Configuration Worksheet. The following sections explain the information you need to record in Part A of the worksheet. If you are viewing this manual on line, you can use the print feature to print a copy of this part of the worksheet.

Figure 2–4: Configuration Worksheet, Part A

Part A: Classical IP			
ATM Hosts file	ATM address:	Host name:	Alias:
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____

Adapter
 Adapter name: _____
 ROM ESIs: _____
 More ESIs: _____
 SDH mode: Yes No
 Flow control: Yes No

Signaling
 Signaling: Yes No
 UNI version: 3.0 3.1
 VC accounting: Yes No

LIS LIS number: _____

ARP ARP: Client Server
 ATM address: _____
 IP address: _____

Adapter name

The device names of the network interfaces. The lta network interface are supported for ATM.

ROM end system identifiers

The ROM end system identifier (ESI) addresses of the adapter that you want to register with the system and the local switch. If you want to register all of the adapter’s ROM ESI addresses, leave this blank.

Depending on the number of address prefixes assigned by the switch, you can create one or more ATM addresses. The driver can control up to 64 ROM ESI addresses, though adapters generally have only a few ROM ESI addresses.

Additional ESIs

Additional ESI addresses that you want to register with the system and the local switch. An ESI address has twelve hexadecimal digits.

Synchronous Digital Hierarchy Mode

If you want to enable Synchronous Digital Hierarchy (SDH) mode on ATM adapters that support both Synchronous Optical Network (SONET) and SDH physical interfaces, check Yes; otherwise, check No.

Flow control

If you want to enable vendor-specific flow control on the adapter, check Yes; otherwise, check No. The adapter must support this type of flow control. DIGITAL adapters and switches support FLOWmaster vendor flow control.

Signaling and ILMI

If you want to enable signaling and the Integrated Layer Management Interface (ILMI) on the interface, check Yes; otherwise, check No. You must enable signaling and ILMI when using Classical IP over SVCs.

UNI version (signaling only)

The signaling version to use on the interface. If you want to use User-Network Interface (UNI) Version 3.0, check 3.0. If you want to use UNI Version 3.1, check 3.1. The default is 3.0.

VC accounting (signaling only)

If you want to enable logging of virtual circuit (VC) releases, check Yes; otherwise, check No.

LIS number

A Logical IP Subnet (LIS) interface number. You can create multiple LIS interfaces on an ATM driver.

ARP server

If you want your system to function as an ARP server, check Server; otherwise, check Client.

Server ATM address (ARP client only)

The ATM address of the ATM ARP server, either a host name or alias that appears in the `/etc/atmhosts` file or a 40-digit ATM End System Address (AESA) with selector byte. The ARP server must also be on the ATM network.

Note

The ATM Forum now calls an NSAP-style address an AESA.

Server IP address (ARP client only)

The IP address of the ATM ARP server machine.

2.2.3.2 LAN Emulation Information

Figure 2–5 shows Part B of the Configuration Worksheet. The following sections explain the information you need to record in Part B of the worksheet. If you are viewing this manual on line, you can use the print feature to print a copy of this part of the worksheet.

Figure 2–5: Configuration Worksheet, Part B

Part B: LAN Emulation			
ATM Hosts file	ATM address:	Host name:	Alias:
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____

Adapter

Adapter name: _____

ROM ESIs: _____

More ESIs: _____

SDH mode: Yes No

Flow control: Yes No

Signaling

Signaling: Yes No

UNI version: 3.0 3.1

VC accounting: Yes No

LANE

ELAN number: _____

LAN name: _____

LES address: _____

MTU size: 1516 4544 9234 18190

Adapter name

The device names of the network interfaces. The `lta` network interfaces are supported for ATM.

ROM end system identifiers

The ROM end system identifier (ESI) addresses of the adapter that you want to register with the system and the local switch. If you want to register all of the adapter's ROM ESI addresses, leave this blank. You must

enter at least one ESI for each emulated LAN (ELAN) you want to configure.

Depending on the number of address prefixes assigned by the switch, you can create one or more ATM addresses. The driver can control up to 64 ROM ESI addresses, though adapters generally have only a few ROM ESI addresses.

Additional ESIs

Additional ESI addresses that you want to register with the system and the local switch. An ESI address has twelve hexadecimal digits.

Synchronous Digital Hierarchy Mode

If you want to enable Synchronous Digital Hierarchy (SDH) mode on ATM adapters that support both Synchronous Optical Network (SONET) and SDH physical interfaces, check Yes; otherwise, check No.

Flow control

If you want to enable vendor-specific flow control on the adapter, check Yes; otherwise, check No. The adapter must support this type of flow control. DIGITAL adapters and switches support FLOWmaster vendor flow control.

Signaling and ILMI

If you want to enable signaling and the Integrated Layer Management Interface (ILMI) on the interface, check Yes; otherwise, check No. You must enable signaling and ILMI when using LAN Emulation.

UNI version (signaling only)

The signaling version to use on the interface. If you want to use User-Network Interface (UNI) Version 3.0, check 3.0. If you want to use UNI Version 3.1, check 3.1. The default is 3.0.

VC accounting (signaling only)

If you want to enable logging of virtual circuit (VC) releases, check YES; otherwise, check NO.

ELAN number

A LAN Emulation Client (LEC) interface unit number.

Virtual LAN name

The name of the emulated LAN to join. The emulated LAN name must already be configured on the ATM switch. If the name is not configured on the ATM switch, the LEC joins the default emulated LAN. Do not specify both a virtual LAN name and a LES address.

LES address

The ATM address of the LAN Emulation Server (LES), either a host name or alias that appears in the `/etc/atmhosts` file or a 40-digit ATM AESA

address with selector byte. If you want the LEC to go directly to the LES and bypass the configuration phase, enter the LES address. Otherwise, the LEC contacts a LAN Emulation Configuration Server (LECS) to obtain a LES address. Do not specify both a virtual LAN name and a LES address.

MTU size

The maximum transmission unit (MTU) size. The following MTU sizes are supported: 1516, 4544, 9234, and 18190. When specified with a virtual LAN name, the emulated LAN must already be configured on the ATM switch to support the specified MTU size. If it is not configured for the specified MTU size, the request is ignored.

2.2.3.3 IP Switching Information

Figure 2–6 shows Part C of the Configuration Worksheet. The following sections explain the information you need to record in Part C of the worksheet. If you are viewing this manual on line, you can use the print feature to print a copy of this part of the worksheet.

Figure 2–6: Configuration Worksheet, Part C

Part C: IP Switching			
Adapter name: _____			
Interface: _____			
SNAP VCI: _____			
Routing: gated <input type="checkbox"/> routed <input type="checkbox"/> static routes <input type="checkbox"/>			
Static routes			
Destination: _____			
Gateway: _____			
Netmask: _____			
Hosts file	Host name:	Internet address:	Alias:
	_____	_____	_____
	_____	_____	_____
	_____	_____	_____
	_____	_____	_____
	_____	_____	_____
	_____	_____	_____
	_____	_____	_____

Adapter name

The device names of the network interfaces. The `lta` network interfaces are supported for ATM.

Interface

The IP switching (`ips`) interface number. If you are using multiple adapters, each adapter is assigned a separate interface number.

SNAP VCI

The Virtual Channel Identifier (VCI) number that Ipsilon Flow Management Protocol (IFMP) uses as the default Subnetwork Attachment Point (SNAP) VCI. The default VCI is 15. This number must match the VCI number that IFMP uses on the destination host or switch associated with the point-to-point interface.

Routing

The method you use to update your internal routing tables. If you use the `gated` daemon, check `gated`. If you use the `routed` daemon, check `routed`. If you use static routes, check `static routes`.

Destination (static routes only)

The IP address of the destination subnetwork.

Gateway (static routes only)

The IP address of the IP controller on the IP switch.

Netmask (static routes only)

The netmask for the destination subnetwork.

Host name

The names of hosts on the subnetwork to be added to the `/etc/hosts` file.

Internet address

The IP addresses of hosts on the subnetwork to be added to the `/etc/hosts` file.

Alias

The aliases, if any, of hosts on the subnetwork to be added to the `/etc/hosts` file.

2.3 Configuring ATM

After you complete the required ATM planning, you configure the ATM software. The following sections provide information on how to configure classical IP, LAN emulation, and IP switching.

2.3.1 Configuring Classical IP

Configuring Classical IP on your host consists of the following steps:

1. Creating PVC mappings on your ATM switch, if you are not going to use SVCs. You also use PVCs when directly connecting two hosts to each other without an ATM switch.
2. Editing the `/etc/atmhosts` file.
3. Adding hosts to the `hosts` database.
4. Running `atmsetup`.
5. Running `netsetup` (SVCs only).
6. Editing the `/etc/atm.conf` file (PVCs only).
7. Running `atmconfig` (PVCs only).

The first step depends on the type of ATM switch you are using. See your ATM switch documentation for this information. The following sections describe steps 2–7.

2.3.1.1 Editing the `/etc/atmhosts` File

You edit the `/etc/atmhosts` file to add the address of the ATM ARP server on your ATM network. The `/etc/atmhosts` file contains mappings of ATM host names to ATM hardware addresses. This file can also contain ATM ESIs and AESAs for specific services on the ATM network. Putting entries in this file enables you to specify the address or service by name instead of specifying a long hexadecimal string.

Entries in the `/etc/atmhosts` file can be one of the following:

- A comment, denoted by a pound sign (#) as the first character
- An address specification

The address specification is similar to that of IP addresses in the `/etc/hosts` file, and has the following format:

```
atm_addr hostname [ alias ... ]
```

The `atm_addr` parameter can consist of ESIs or AESAs.

The following table lists the address type and the number of hexadecimal address digits required for each type:

Address Type	Number of Address Digits
ESI	Twelve hexadecimal digits
AESA	Thirty-eight hexadecimal digits
AESA with selector byte	Forty hexadecimal digits

The *hostname* parameter can contain any printable character.

The following example shows entries in the `/etc/atmhosts` file:

```
08002b2fe740                               myhost.esi  1
47840f01020300002122313208002b2fe740    myhost     2
47840f01020300002122313208002b2fe7403a  myhost.ip  3
```

- 1 Specifies an ESI to use in registering `myhost` with the switch.
- 2 Specifies the AESA of `myhost`. This is the network prefix and the ESI, and is the address that the network knows.
- 3 Specifies the AESA with selector byte of a service on `myhost` for the DIGITAL UNIX implementation of RFC 1577, *Classical IP and ARP over ATM*.

Note

By default, the `atmhosts` file contains an entry for PVCs. Do not delete or modify this entry.

2.3.1.2 Adding Hosts to the hosts Database

You add the IP addresses for all ATM hosts that will be on any Logical IP Subnet (LIS) to which the host will connect to the `hosts` database. Make sure you have the IP addresses for the local host and the ATM ARP server. Depending on your environment, host names and addresses can be in the local `/etc/hosts` file or in one of the files distributed with the Berkeley Internet Name Domain (BIND) system or Network Information System (NIS).

You can enter these IP addresses in the `/etc/hosts` file either by editing the file itself or by running the Network Configuration application of the Common Desktop Environment (CDE) Application manager. See *Network Administration* for information on the Network Configuration application, BIND, and NIS.

2.3.1.3 Running atmsetup

To run the `atmsetup` script and to configure Classical IP on your system, log in as root and do the following:

1. Invoke the `atmsetup` script by entering the following command:

```
# /usr/sbin/atmsetup
```

The ATM Setup menu appears.

2. Enter the number of the adapter that you want to configure.
3. Enter 1 (configure) to configure the adapter for Classical IP.
4. Enter 1 (Driver) to configure the driver module.
5. Enter 1 (configure) to configure the driver.
6. Indicate whether you want the driver to come on line and initiate low-level communications with the ATM switch when the system is booted.
7. Indicate whether you want to register all of the driver's ESIs. If you answer `y` (the default), all of the adapter's ROM ESI addresses are registered. If you answer `n`, you are prompted to select the ESIs from a list and to enter the number of each, separated by commas. For example, 1,3,5.
8. Indicate whether you want to register any additional ESIs. If you answer `y`, you are prompted for the ESI address.
9. Indicate whether you want to enable SDH mode. If you do, enter `y`; otherwise, enter `n`.
10. Indicate whether you want to enable FLOWmaster vendor flow control. If you do, enter `y`; otherwise, enter `n`.
11. Enter 2 (Signaling) to configure the signaling module.
12. Enter 1 (configure) to configure signaling.
13. Indicate whether you want to enable signaling and ILMI on the interface.

If you answer `y` (the default), enter the UNI version you want to use. Press Enter to accept the default (Version 3.0); otherwise, enter 3.1. Then, indicate whether you want to log VC activity.
14. Enter 3 (ATMIP) to configure the ATMIP (Classical IP) module.
15. Enter 1 (configure) to configure ATMIP.
16. Enter 1 to add a LIS.
17. Enter the number of the LIS you want to add.

18. Indicate whether your system is to act as an ARP client or an ARP server. Enter `c` for client or `s` for server.
19. If the system is to be an ARP client, enter the ARP server's ATM address or alias and press Enter. Then, enter the ARP server's IP address and press Enter.
20. To add another LIS, enter `1`; otherwise, enter `6` to indicate that you do not want to add more LISs. Then, enter `5` to indicate that you are finished with the ATMIP module setup. Then, enter the last number displayed (done) to indicate that you are finished with the adapter setup.
21. If you want to start the ATM components immediately from the `atmsetup` script, enter `y`; otherwise, enter `n`. If you enter `n`, you must reboot the system to start the ATM components with the new values.

2.3.1.4 Running netsetup (SVCs only)

After you run the `atmsetup` script and start the ATM components (either from within the `atmsetup` script or by rebooting the system), you run the `/sbin/netsetup` script to configure the `lis` interfaces. See `netsetup(8)` for more information.

2.3.1.5 Editing the `/etc/atm.conf` File (PVCs only)

The basic Classical IP configuration process creates switched virtual circuits (SVCs). If you want to use permanent virtual circuits (PVCs), you must manually edit the `/etc/atm.conf` file after you run `atmsetup`. After you install the ATM subsets, the ATM configuration file contains the following sample PVC entries:

```

:
# print Waiting to join ATM Network... 1
# wait state=up driver=lta0 2
# sleep 3 3
##
## *****
## This sequence shows how to configure a Classical IP interface over a
## Permanent Virtual Circuit (PVC).
## *****
##
## Create the Logical IP Subnet (LIS)
# print Configuring Classical IP interface lis0 on VPI=0 VCI=50 ... 4
# run /usr/sbin/atmarp -c lis=0 driver=lta0 5
##
## Set the IP address for the LIS. A PVC LIS cannot be configured through
## the netsetup command because it does not exist until the atmarp command
## above is executed.
# run ifconfig lis0 10.128.64.7 netmask 255.255.255.0 up 6
#
## Set up the PVC. The same PVC must also have been configured on the switch.
## Note: The selector value will be different for each LIS (use "atmconfig

```



```

## bindlist" to see the bindings and selectors).
# +pvc driver=lta0 converge=atmip vpi=0 vci=50 selector=0 [7]
##
## If the remote host does not support RFC1577, the following additional
## commands are necessary.
##
## Create a permanent ATM ARP entry for the remote host's IP address
# atmarp -s PVC 10.128.64.xx permanent [8]
##
## Associate the VPI/VCI specified above with the IP address
# atmarp -v +vc 0 50 10.128.64.xx [9]
##
:
:

```

- [1] Remove the comment character (#) to enable printing of the message.
- [2] Waits for the driver to complete initialization with the switch, delaying startup until the interface is up.

Remove the comment character (#) and edit this entry as needed. Add entries for each additional interface installed on the system, incrementing the interface number. For example, if another DGLTA adapter is installed on the system, the entry would be as follows:

```
wait state=up driver=lta1
```

- [3] Suspends execution for three seconds to allow the driver to complete initialization.

Remove the comment character (#) and edit this entry as needed.

- [4] Creates and attaches the LIS.

Remove the comment character (#) and add entries for each LIS connected to the system, incrementing the LIS number.

- [5] Associates an IP address with the LIS.

The LIS entry has the following format:

```
run ifconfig lisen atmipaddr netmask ipnetmask up
```

The *n* is the LIS number, *atmipaddr* is the IP address of the local machine on the ATM LIS, and *ipnetmask* is the LIS netmask to use.

Remove the comment character (#) and edit this entry as needed. Add entries for each LIS connected to the system, incrementing the LIS number.

- [6] Sets up a PVC for a host on the LIS. Each entry has the following format:

```
+pvc driver=driver converge=atmip vpi=vp_num vci=vc_num selector=num
```

The *driver* option is the driver name associated with the LIS, the *vp_num* option is the VPI of the PVC, the *vc_num* option is the VCI of the PVC configured on the switch, and the *num* option is the LIS number. You must specify the *selector* parameter.

Remove the comment character (#) and edit this entry as needed. Add entries for each additional PVC that you want to set up.

- 7 Depending on your network topology and the number and configuration of LISs in your network, you might need to add static routes to other hosts if either of the following conditions are true:

- Your system is one hop away from another host (via a gateway) over an ATM connection
- Your system is more than one hop away from another host on a network that supports multicast messages (for example, Ethernet, FDDI, Token Ring)

To add a static route to the routing tables, add entries immediately after this comment line. These entries have the following format:

```
run /usr/sbin/route add -net destination gateway -netmask mask
```

The *destination* argument is the IP address of the destination network and the *gateway* argument is the address of the local LIS.

Add entries for each additional host to which your system needs to communicate. See `route(8)` for more information.

- 8 Creates a permanent entry in the ATM ARP table. Each entry has the following format:

```
/usr/sbin/atmarp -s atm_address ip_address permanent
```

The *atm_address* argument specifies the ATM address (full 20-byte AESA or host name) for the entry and the *ip_address* argument specifies the IP address for the corresponding ATM address. The `permanent` argument means that the entry is not subject to the RFC 1577 aging rules. PVCs use the host name `PVC`.

Remove the comment character (#) and edit this entry as needed. Add entries for each additional host that does not support RFC 1577.

- 9 Associates a VPI and VCI with a permanent IP address for a host that does not support RFC 1577. Each entry has the following format:

```
/usr/sbin/atmarp -v +vc vpi_number vci_number hostname
```

The *vpi_number* argument specifies the virtual path identifier (VPI) for the virtual circuit, the *vci_number* argument specifies the virtual channel identifier (VCI) for the virtual circuit, and *hostname* argument specifies the host name assigned to the ARP entry.

Remove the comment character (#) and edit this entry as needed. Add entries for each additional host that does not support RFC 1577.

2.3.1.6 Running `atmconfig` (PVCs only)

After you are done editing the `/etc/atm.conf` file, enter the following command to start and configure the ATM subsystem:

```
# atmconfig source
```

If you have multiple configuration files, you can execute a specific file by entering the following command:

```
# atmconfig source file=file_name
```

See `atmconfig(8)` for more information.

Note

You must invoke the `atmconfig` command to process the commands in the `/etc/atm.conf` file each time the system is rebooted. You can do this manually or by running a user-supplied script in the `/etc/inin.d` directory.

After the PVC is configured, verify the configuration by issuing the `atmarp -a` command. Output similar to the following appears if the PVC is configured:

```
# atmarp -a
Number of entries : 1

IP Address :   atm66 (16.142.128.66)
ATM Address :   PVC
Flags :        Complete Permanent
VCs :          vpi    vci    VC Type
               ---    ---    -
                   0     999    PVC
```

2.3.2 Configuring LAN Emulation

Configuring LAN emulation on your host consists of the following steps:

1. Editing the `/etc/atmhosts` file
2. Adding hosts to the `hosts` database
3. Running `atmsetup`
4. Running `netsetup`

The following sections describe these steps.

2.3.2.1 Editing the `/etc/atmhosts` File

You edit the `/etc/atmhosts` file only if you want to specify a LAN Emulation Server (LES) address on your ATM network. The

`/etc/atmhosts` file contains mappings of ATM host names to ATM hardware addresses. This file can also contain ATM ESIs and AESAs for specific services on the ATM network.

Entries in the `/etc/atmhosts` file can be one of the following:

- A comment, denoted by a pound sign (#) as the first character
- An address specification

The address specification is similar to that of IP addresses in the `/etc/hosts` file, and has the following format:

```
atm_addr hostname [ alias ... ]
```

The `atm_addr` parameter can consist of ESIs or AESAs.

The following example shows entries in the `/etc/atmhosts` file:

```
08002b2fe740                               myhost.esi 1
47840f01020300002122313208002b2fe740    myhost 2
47840f01020300002122313208002b2fe7403a    myhost.ip 3
```

- 1 Specifies an ESI to use in registering `myhost` with the switch.
- 2 Specifies the AESA of `myhost`. This is the network prefix and the ESI, and is the address that the network knows.
- 3 Specifies the AESA with selector byte of a service on `myhost` for the DIGITAL UNIX implementation of RFC 1577, *Classical IP and ARP over ATM*.

Note

By default, the `atmhosts` file contains an entry for PVCs. Do not delete or modify this entry.

2.3.2.2 Adding Hosts to the hosts Database

You add the IP addresses for all ATM hosts that will be on any emulated LAN (ELAN) to which the host will connect to the `hosts` database. Make sure you have the IP addresses for the local host. Depending on your environment, host names and addresses can be in the local `/etc/hosts` file or in one of the files distributed with the Berkeley Internet Name Domain (BIND) system or Network Information System (NIS).

You can enter these IP addresses in the `/etc/hosts` file either by editing the file itself or by running the Network Configuration application of the Common Desktop Environment (CDE) Application manager. See *Network Administration* for information on the Network Configuration application, BIND, and NIS.

2.3.2.3 Running atmsetup

To run the `atmsetup` script and to configure LAN emulation on your system, log in as root and do the following:

1. Invoke the `atmsetup` script by entering the following command:

```
# /usr/sbin/atmsetup
```

The ATM Setup menu appears.

2. Enter the number of the adapter that you want to configure.
3. Enter 1 (configure) to configure the adapter for LAN emulation.
4. Enter 1 (Driver) to configure the driver module.
5. Enter 1 (configure) to configure the driver.
6. Indicate whether you want the driver to come on line and initiate low-level communications with the ATM switch when the system is booted.
7. Indicate whether you want to register all of the driver's ESIs. If you answer `y` (the default), all of the adapter's ROM ESI addresses are registered. If you answer `n`, you are prompted to select the ESIs from a list and enter the number of each, separated by commas. For example, 1,3,5.
8. Indicate whether you want to register any additional ESIs. If you answer `y`, you are prompted for the ESI address.
9. Indicate whether you want to enable SDH mode. If you do, enter `y`; otherwise, enter `n`.
10. Indicate whether you want to enable FLOWmaster vendor flow control. If you do, enter `y`; otherwise, enter `n`.
11. Enter 2 (Signaling) to configure the signaling module.
12. Enter 1 (configure) to configure signaling.
13. Enter `y` to indicate that you want to enable signaling and ILMI on the interface. Then, enter the UNI version you want to use. Press Enter to accept the default (Version 3.0); otherwise, enter 3.1. Then, indicate whether you want to log VC activity.
14. Enter 4 (LANE) to configure the LANE (LAN Emulation) module.
15. Enter 1 (configure) to configure LANE.
16. Enter 1 to add an ELAN.
17. Indicate whether you want to join a specific emulated LAN. If you do, enter `y`. Then, enter the name of the emulated LAN you want to join and go to step 19.

If you do not want to join a specific emulated LAN, enter `n`.

18. If you answered `n` to the previous question, indicate whether you want to specify a LAN Emulation Server (LES). If you answer `y`, enter the LES address. If you answer `n`, you will use the LAN Emulation Configuration Server (LECS) and join the default emulated LAN.
19. Indicate whether you want to specify an MTU size other than the default 1516. If you answer `y`, enter the MTU size.
20. To add another ELAN, enter `1`; otherwise, enter `6` to indicate that you do not want to add more ELANs. Then, enter `5` to indicate that you are finished with the LANE module setup. Then, enter the last number displayed (done) to indicate that you are finished with the adapter setup.

Note

You can join an emulated LAN on a switch only once for each adapter; do not join the same ELAN multiple times from the same adapter. If you want to join the same ELAN on the same switch, you must install another adapter and join the ELAN from it.

21. If you want to start the ATM components immediately from the `atmsetup` script, enter `y`; otherwise, enter `n`. If you enter `n`, you must reboot the system to start the ATM components with the new values.

2.3.2.4 Running `netsetup`

After you run the `atmsetup` script and start the ATM components (either from within the `atmsetup` script or by rebooting the system), you run the `/sbin/netsetup` script to configure the `elan` interfaces. See `netsetup(8)` for more information.

2.3.3 Configuring IP Switching

Configuring IP switching on your host consists of the following steps:

1. Editing the `/etc/hosts` file
2. Running `atmsetup`
3. Editing the `/etc/atm.conf` file
4. Running `atmconfig`

The following sections describe these steps.

2.3.3.1 Editing the /etc/hosts File

You edit the `/etc/hosts` file to add the IP addresses for each IP switching subnetwork to which the host will connect. For each subnet, add a pair of IP addresses for each end of the point-to-point link (host side and IP controller side), the IP address of the subnet, and the broadcast address of the subnet. For example, an `/etc/hosts` file for the configuration in Figure 2-3 is as follows:

```
# IP Switching subnet A
16.1.1.4   atm4-net
16.1.1.5   hosta.corp.com           hosta           atm5
16.1.1.6   ipsctrlhosta.corp.com   ipsctrlhosta   atm6
16.1.1.7   atm7-broadcast
# IP Switching subnet B
16.1.1.0   atm0-net
16.1.1.1   ipsctrlhostb.corp.com   ipsctrlhostb   atm1
16.1.1.2   hostb.corp.com          hostb           atm2
16.1.1.4   atm4-broadcast
# IP Switching subnet C
16.1.1.8   atm8-net
16.1.1.9   ipsctrlhostc.corp.com   ipsctrlhostc   atm9
16.1.1.10  ipgwy.corp.com           ipgwy           atm10
16.1.1.11  atm11-broadcast
```

You can enter these IP addresses either by editing the `/etc/hosts` file or by running the Network Configuration application of the Common Desktop Environment (CDE) Application manager. See *Network Administration* for information on the Network Configuration application.

2.3.3.2 Running atmsetup

To run the `atmsetup` script and to configure the adapter or adapters on your system, log in as root and do the following:

1. Invoke the `atmsetup` script by entering the following command:

```
# /usr/sbin/atmsetup
```

The ATM Setup menu appears.

2. Enter the number of the adapter that you want to configure.
3. Enter 1 (configure) to configure the adapter.
4. Enter 1 (Driver) to configure the driver module.
5. Enter 1 (configure) to configure the driver.
6. Indicate whether you want the driver to come on line and initiate low-level communications with the ATM switch when the system is booted.

7. Enter *y* to indicate that you want to register all of the driver's ROM ESIs.
8. Enter *n* to indicate that you do not want to register any additional ESIs.
9. Enter *n* to indicate that you want to enable SONET mode.
10. Enter *n* to indicate that you do not want to enable vendor flow control.
11. Enter 5 to indicate that you are finished with the driver module setup. Then, enter the last number displayed (done) to indicate that you are finished with the adapter setup.
12. If you want to start the ATM components immediately from the `atmsetup` script, enter *y*; otherwise, enter *n*. If you enter *n*, you must reboot the system to start the ATM components with the new values.

2.3.3.3 Editing the `/etc/atm.conf` File

After you install the ATM subsets, the default ATM configuration file, `/etc/atm.conf`, contains entries for Classical IP over PVCs and for IP switching. Use the following entries for configuring IP switching:

```

:
# print Waiting to join ATM network ... 1
# wait state=up driver=lta0 2
# sleep 3 3
:
:
##
## *****
## This sequence shows how to configure an IP Switching interface
## *****
##
## Create an ips0 interface. Override default SNAP VCI to interoperate
## with the Gigaswitch Rev 2.x software.
# print Configuring IP Switching interface ips0 ...
# run /usr/sbin/atmifmp attach ips=0 driver=lta0 snapvci=32 4
## Set the IP address for the interface. An IP Switching interface cannot
## be configured through the netsetup command because it does not exist
## until the above atmifmp command is executed.
# run /usr/sbin/ifconfig ips0 up 10.142.128.133 10.142.128.134 netmask 255.255.255.252 5
##
## If not running a routing daemon (routed, etc) one or more static routes
## may be necessary.
# run /usr/sbin/route add -net 10.142.128.0 10.142.128.134 -netmask 255.255.255.0 6
exit

```

- 1 Remove the comment character (#) to enable printing of the message.
- 2 Waits for the driver to complete initialization with the switch, delaying startup until the interface is up.

Remove the comment character (#) and edit this entry as needed. Add entries for each additional interface installed on the system,

incrementing the interface number. For example, if another DGLTA adapter is installed on the system, add the following entry:

```
wait state=up driver=lta1
```

- 3 Suspend execution for three seconds to allow the driver to complete initialization.

Remove the comment character (#) and edit this entry as needed.

- 4 Creates and attaches the `ips0` interface, and overrides the default SNAP VCI to interoperate with the DIGITAL GIGAswitch switch.

Remove the comment character (#) and edit this entry as needed. Add entries for each `ips` interface, incrementing the `ips` number.

Note

The default SNAP VCI is 15. This number must match the VCI number that IFMP uses on the switch associated with the point-to-point interface.

- 5 Associates an IP address with the `ips` interface.

The `ips` entry has the following format:

```
run ifconfig ips0 atmipaddr addr netmask ipnetmask up
```

The `atmipaddr` is the IP address of the local machine, `addr` is the IP address of the IP controller, and `ipnetmask` is the netmask to use.

Remove the comment character (#) and edit this entry as needed. Add entries for each `ips` interface on the system, incrementing the `ips` number.

If you were configuring IP switching on Host A in Figure 2–3, you would create the following entry:

```
run /usr/sbin/ifconfig ips0 up 16.1.1.5 16.1.1.6 netmask 255.255.255.252
```

- 6 Adds a static route to the subnetwork 10.142.128.0.

Depending on your network topology and the number of interfaces on your host, you might need to add static routes to other hosts if your system has multiple interfaces and the default route is to another gateway on another network.

Remove the comment character (#) and edit this entry as needed. Add entries for each additional route to add to the routing tables. These entries have the following format:

```
run /usr/sbin/route add -net destination gateway -netmask mask
```

The *destination* argument is the IP address of the destination subnetwork, and the *gateway* argument is the address of the IP controller on the IP switch. See `route(8)` for additional information.

If you were configuring IP switching on Host A in Figure 2–3 and you wanted to route all traffic on all 16.1.1 networks through the IP switch, you would create the following static route to the IP controller on the switch:

```
run /usr/sbin/route add -net 16.1.1.0 16.1.1.6 -netmask 255.255.255.0
```

Using the Class Inter-Domain Routing (CIDR) format, the entry would be as follows:

```
run /usr/sbin/route add -net 16.1.1/24 16.1.1.6
```

2.3.3.4 Running `atmconfig`

After you are done editing the `/etc/atm.conf` file, enter the following command to start and configure the ATM subsystem:

```
# atmconfig source
```

If you have multiple configuration files, you can execute a specific file by entering the following command:

```
# atmconfig source file=file_name
```

See `atmconfig(8)` for more information.

2.4 Managing the ATM Environment

Managing the ATM environment consists of the following tasks:

- Managing ATM networking and displaying information about ATM networks
- Managing the signaling module
- Managing the Classical IP environment
- Managing the LAN Emulation environment
- Managing IP switching

The following sections describe each of these tasks.

2.4.1 Managing ATM Networking and Displaying Information About ATM Networks

To manage ATM networking and to display information about the ATM networks, you use the `atmconfig` command. The command controls only

the base ATM modules and device drivers; it does not control specific convergence modules or signaling protocols. You can use the `atmconfig` command to do the following:

- Enable and disable device drivers
- Create and destroy PVCs
- Destroy SVCs
- Create and destroy ESIs
- Display the currently active VCs and driver status
- Process configuration batch files

See `atmconfig(8)` for more information.

2.4.2 Managing the Signaling Module

To manage ATM UNI signaling on the end system, you use the `atmsig` command. The `atmsig` command allows you to:

- Display state information about the signaling module
- Disable and enable the ILMI and signaling
- Read and modify the various timer values and statistics for Q.SAAL and Q.93B (2931)

The signaling module is associated with a specified interface at all times, which is identified by the driver name. If the interface is disabled, the signaling module is also disabled. The signaling module must be enabled again when the interface is brought back on line.

See `atmsig(8)` for more information.

2.4.3 Managing the Classical IP Environment

To manage Classical IP on an end system, you use the `atmarp` command. The `atmarp` command allows you to:

- Create a logical IP subnet (LIS) interface
- Create and delete entries in the ATM ARP table
- Display entries in the ATM ARP table
- Toggle the permanent flag for entries
- Display the local host's ATM configuration status
- Create and remove an association between an established VC and a remote IP entity that does not support Classical IP

See `atmarp(8)` for more information.

2.4.4 Managing the LAN Emulation Environment

Managing the LAN emulation environment consists of the following tasks:

- Managing LAN Emulation Clients (LECs)
- Displaying the LAN Emulation Address Resolution Protocol (LE-ARP) table

The following sections describe these tasks.

2.4.4.1 Managing LAN Emulation Clients

To manage LAN Emulation Clients (LECs), you use the `atmelan` command. The `atmelan` command allows you to:

- Create and configure LAN Emulation Clients (LEC) as network interfaces
- Display counters, parameters, and the state of each LEC

See `atmelan(8)` for more information.

2.4.4.2 Displaying the LE-ARP Table

To display the LE-ARP table for each `elan` interface, you use the `learp` command. The command displays the address mappings for the emulated LAN. Each entry consists of the Media Access Control (MAC) address, state, ATM address, and flags. See `learp(8)` for more information.

2.4.5 Managing IP Switching

To manage IP switching on an end system, you use the `atmifmp` command. The `atmifmp` command allows you to:

- Enable and disable IP switching
- Display IP switching configuration
- Display or clear IP switching statistics
- Display IP switching flow information

See `atmifmp(8)` for more information.

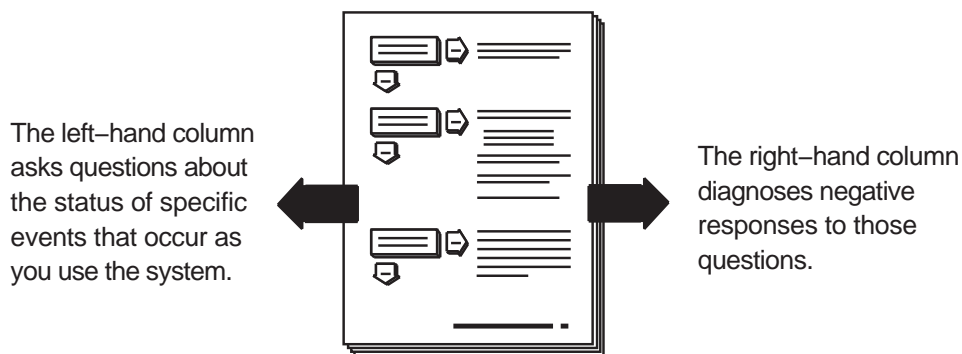
2.5 Solving Problems

This section contains a diagnostic map to help you solve problems that might occur when you use the ATM network services. Use this section,

along with the appropriate DIGITAL documentation, to solve as many problems as possible at your level.

2.5.1 Using the Diagnostic Map

ATM network problems can occur for a number of reasons. The diagnostic map in this section should help you isolate the problem. The following figure explains how to use the diagnostic map:



After you isolate the problem, the map refers you to other sections for instructions on using the various problem-solving tools and utilities. The map also refers you to other manuals for complete diagnostic information for particular devices and software products.

You could experience problems that are not documented in this manual when you use ATM network software with other layered products. Each layered product has its own manual or set of manuals that might describe solutions not provided here.

2.5.2 Getting Started

Before you start problem solving, ensure that communications hardware is ready for use. Verify the following:

- The system's physical connections are properly installed. See the documentation for your system and communications hardware device.
- Event logging is enabled to monitor network events. See *System Administration* for information on starting event logging and for descriptions of event messages.

Also check the product release notes for up-to-date information on known problems.

2.5.3 Solving ATM Problems

ATM subsets installed?

NO

Verify that the ATM subsets are installed. Enter the following command:

```
# setld -i | grep OSFATM
```

The following messages should appear:

```
OSFATMnnn installed ATM Commands
(Network-Server/Communications)
OSFATMBINnnn installed ATM Kernel
Modules (Kernel Build Environment)
OSFATMBINCOMnnn installed ATM Kernel
Header and Common Files
(Kernel Build Environment)
OSFATMBINOBJECTnnn installed ATM Kernel
Objects (Kernel Software Environment)
```

If the subsets are not installed, install them by using the `setld` command. See *System Administration* for information on installing the subset.

NOTE

You do not have to install the `OSFATMBINOBJECT` subset.

ATM configured in kernel?

NO

Verify that the ATM support you want is configured in the kernel. Enter the following command:

```
# sysconfig -q atm
```

If nothing is displayed, ATM is not configured in the kernel. Reconfigure the kernel with the ATM option and additional ATM options as needed. See Section 2.2.2 for a list of ATM kernel options and for information on reconfiguring the kernel.

If ATM is configured in the kernel, use the `sysconfig -q` command to verify that other ATM kernel options are configured. Reconfigure the kernel with additional options as needed.

ATM driver configured?

NO

Verify that the driver is configured by issuing the `atmconfig drvlist` command. If the driver is configured, output similar to the following appears:

```
Name: lta0      Type: STS-3      State: UP
Driver ID: 1   ESIs: 8   PPAs: 9   VCs: 6
```

If an entry for the driver does not exist, use the `genvmunix` kernel to reboot the system and run `doconfig` to build a kernel with the required driver.
If the driver state is not UP, run the `atmsetup` utility for the specific ATM service you want. See Section 2.3.1.3, Section 2.3.2.3, and Section 2.3.3.2 for information on configuring the driver for CLIP, LANE, and IP switching, respectively.

YES

Go to Section 2.5.3.1 for Classical IP, go to Section 2.5.3.2 for LAN Emulation, or go to Section 2.5.3.3 for IP switching.

2.5.3.1 Solving CLIP Problems

Signaling configured? (SVCs only)

NO

Verify that signaling is configured by issuing the following command:

```
# atmsig status driver=driver_name
```

If no UNI version number is displayed or the ILMI state is Unknown, run the `atmsetup` utility and configure signaling. See Section 2.3.1.3 for information.

YES

lis interfaces created?

NO

Verify that the CLIP `lis` interfaces are created. Enter the following command:

```
# atmarp -h
```

If a `lis` interface is created, the status of all created LISs and whether the host is an ARP client or ARP server appears.
If no LISs are created, run the `atmsetup` utility and configure CLIP. See Section 2.3.1.3 for more information.

YES

lis interfaces
configured?



Verify that a `lis` interface is configured. Enter the following command:

```
# ifconfig lisx
```

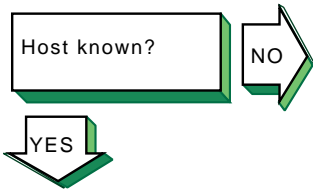
If a `lis` interface is configured, output similar to the following appears:

```
lis0: flags=c23<UP,BROADCAST,NOTRAILERS,MULTICAST,SIMPLEX>  
      inet 10.140.120.52 netmask ffffffff broadcast 10.140.120.255  
      ipmtu 1500
```

For SVCs, if a `lis` interface is not configured, run the `netsetup` utility and configure one. See Section 2.3.1.4 for more information.

For PVCs, if a `lis` interface is not configured, edit the `atm.conf` file and run the `atmconfig` command. See Section 2.3.1.5 for more information.





If a remote host is not known, the following message appears:

unknown host

Complete the following steps:

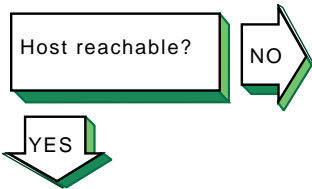
1. Check whether the user is using a valid host name to reach the remote host.
2. Check if the remote host is in another name domain and whether the user specified the full domain name.
3. If your site uses the BIND name service for name-to-address translation, check the `/etc/svc.conf` file to see if `bind` is specified as a service for the `hosts` database entry. If it is not, edit the file and add it.

Also, verify that the BIND service has information about the remote host. See the solutions for solving BIND client problems in *Network Administration*.

4. If your site uses NIS name service for name-to-address translation, check the `/etc/svc.conf` file to see if `nis` is specified as a service for the `hosts` database entry. If it is not, edit the file and add it.

Also, verify that the NIS service has information about the remote host. See the solutions for solving NIS client problems in *Network Administration*.

5. If your `/etc/svc.conf` file lists `local` as the only name-to-address translation mechanism, the `/etc/hosts` file does not have information on the remote host. See *System Administration* for more information.

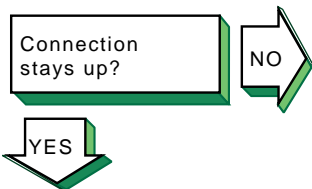


If a remote host is not reachable, the following message appears:

host is unreachable

Complete the following steps:

1. Check the cabling between the local host and the switch.
2. Verify that there is network connectivity to the IP controller on the switch by using the `ping` command. If the command fails, possible reasons include: the `ifconfig` command parameters are wrong, or the IP controller is down or has an interface problem. Contact the switch administrator.
3. Verify that there is network connectivity to the target remote host by using the `ping` command. If the command fails, use the `traceroute` command to check the route to the remote host.



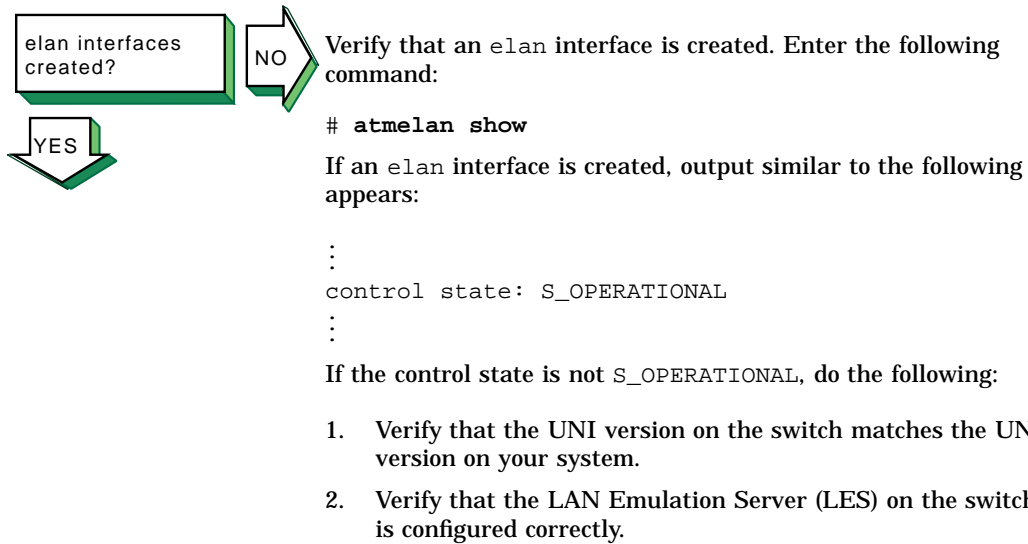
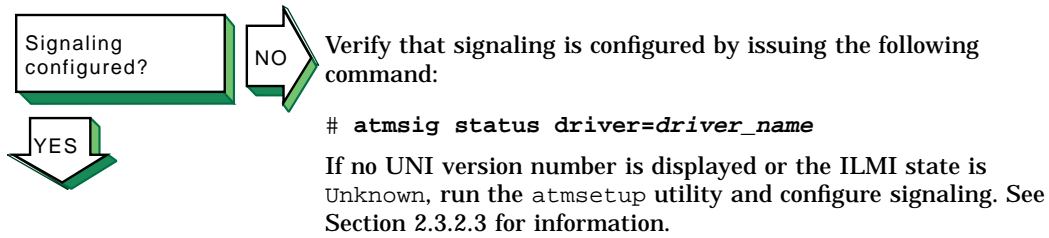
If the connection terminates abnormally, complete the following steps:

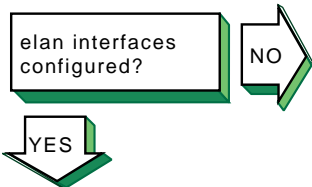
1. Test the network to determine whether the problem is on the local host, remote host, or a host on the path between the two. See *Network Administration* for more information on testing the network.
2. Once you have identified the host with the problem, do the following:
 - a. Verify that the broadcast address and address mask for the local host are properly set up in the `/etc/rc.config` file.
 - b. Confirm that the network device is properly configured.
 - c. Make sure the local host's `hosts` database has the correct IP addresses.
 - d. Make sure the cabling from the local host to the network is intact and properly connected.
 - e. If connected over a local area network (LAN), check the Address Resolution Protocol (ARP) entries and LAN connections.



Problem still exists?
Report your problem to
DIGITAL. See the *Network
Administration* guide.

2.5.3.2 Solving LANE Problems





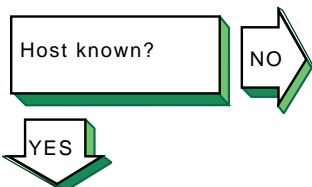
Verify that an elan interface is configured. Enter the following command:

```
# ifconfig elanx
```

If an elan interface is configured, output similar to the following appears:

```
elan0: flags=c23<UP,BROADCAST,NOTRAILERS,MULTICAST,SIMPLEX>
       inet 10.140.120.52 netmask fffffff0 broadcast 10.140.120.255
       ipmtu 1500
```

If an elan interface is not configured, run the `netsetup` utility and configure one. See Section 2.3.2.4 for more information.



If a remote host is not known, the following message appears:

```
unknown host
```

Complete the following steps:

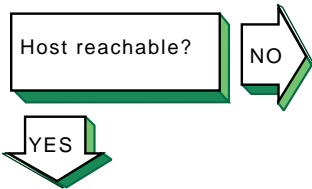
1. Check whether the user is using a valid host name to reach the remote host.
2. Check if the remote host is in another name domain and whether the user specified the full domain name.
3. If your site uses the BIND name service for name-to-address translation, check the `/etc/svc.conf` file to see if `bind` is specified as a service for the `hosts` database entry. If it is not, edit the file and add it.

Also, verify that the BIND service has information about the remote host. See the solutions for solving BIND client problems in *Network Administration*.

4. If your site uses NIS name service for name-to-address translation, check the `/etc/svc.conf` file to see if `nis` is specified as a service for the `hosts` database entry. If it is not, edit the file and add it.

Also, verify that the NIS service has information about the remote host. See the solutions for solving NIS client problems in *Network Administration*.

5. If your `/etc/svc.conf` file lists `local` as the only name-to-address translation mechanism, the `/etc/hosts` file does not have information on the remote host. See *System Administration* for more information.

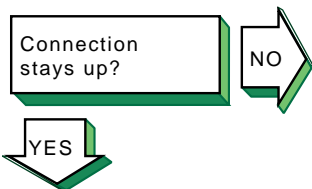


If a remote host is not reachable, the following message appears:

```
host is unreachable
```

Complete the following steps:

1. Check the cabling between the local host and the switch.
2. Verify that the addresses on the link are correct by using the `ifconfig elanx` command.
3. Verify that there is network connectivity to the target remote host by using the `ping` command. If the command fails, use the `traceroute` command to check the route to the remote host.



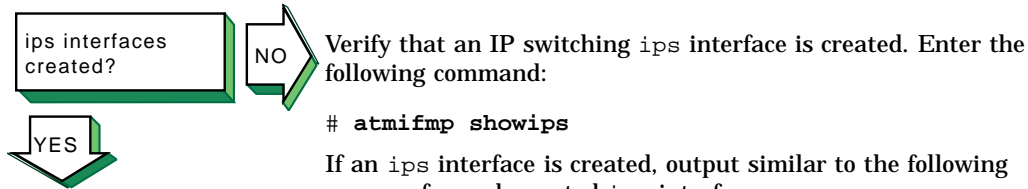
If the connection terminates abnormally, complete the following steps:

1. Test the network to determine whether the problem is on the local host, remote host, or a host on the path between the two. See *Network Administration* for more information on testing the network.
2. Once you have identified the host with the problem, do the following:
 - a. Verify that the broadcast address and address mask for the local host are properly set up in the `/etc/rc.config` file.
 - b. Confirm that the network device is properly configured.
 - c. Make sure the local host's `hosts` database has the correct IP addresses.
 - d. Make sure the cabling from the local host to the network is intact and properly connected.
 - e. If connected over a local area network (LAN), check the Address Resolution Protocol (ARP) entries and LAN connections.



Problem still exists?
Report your problem to
DIGITAL. See the *Network
Administration* guide.

2.5.3.3 Solving IP Switching Problems



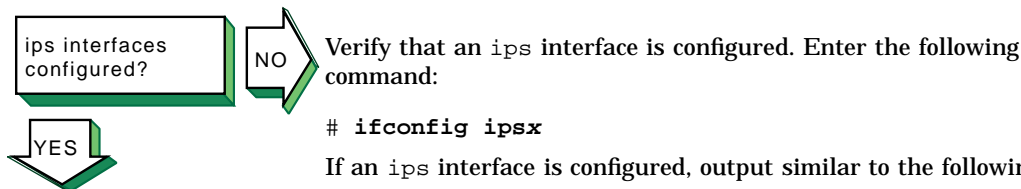
atmifmp showips

If an `ips` interface is created, output similar to the following appears for each created `ips` interface:

```
ips0:
  Attached to driver lta0
  Default (SNAP) VC = 32
  IP Traffic VC = 1850 (Unused - peer does
    not support Flow Type 0)
  Min Tx VC = 1
  Max Tx VC = 2048
  Min Rx VC = 1
  Max Rx VC = 2048
  Driver Min Tx VC = 1
  Driver Max Tx VC = 2048
  Driver Min Rx VC = 1
  Driver Max Rx VC = 2048
  Peer does not support Flow Type 0
```

This example shows that the `ips0` interface has been created and is attached to driver `lta0`.

If no `ips` interfaces are found, create one or more `ips` interfaces. See Section 2.3.3 for more information.



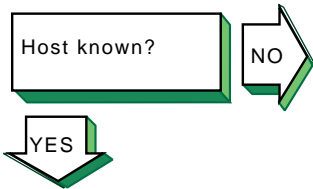
ifconfig ipsx

If an `ips` interface is configured, output similar to the following appears:

```
ips0: flags=4d1<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>
  inet 16.142.128.129 --
  > 16.142.128.130 netmask ffffffff ipmtu 1500
```

The example shows that the interface is up and running and that addresses are configured for each end of the point-to-point link.

If an `ips` interface is not configured, edit the `/etc/atm.conf` file and issue the `atmconfig` command. See Section 2.3.3.3 and Section 2.3.3.4 for more information.



If a remote host is not known, the following message appears:

```
unknown host
```

Complete the following steps:

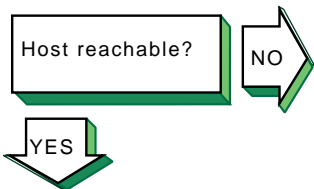
1. Check whether the user is using a valid host name to reach the remote host.
2. Check if the remote host is in another name domain and whether the user specified the full domain name.
3. If your site uses the BIND name service for name-to-address translation, check the `/etc/svc.conf` file to see if `bind` is specified as a service for the `hosts` database entry. If it is not, edit the file and add it.

Also, verify that the BIND service has information about the remote host. See the solutions for solving BIND client problems in *Network Administration*.

4. If your site uses NIS name service for name-to-address translation, check the `/etc/svc.conf` file to see if `nis` is specified as a service for the `hosts` database entry. If it is not, edit the file and add it.

Also, verify that the NIS service has information about the remote host. See the solutions for solving NIS client problems in *Network Administration*.

5. If your `/etc/svc.conf` file lists `local` as the only name-to-address translation mechanism, the `/etc/hosts` file does not have information on the remote host. See *System Administration* for more information.

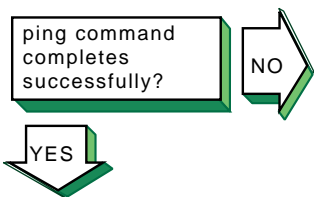


If a remote host is not reachable, the following message appears:

```
host is unreachable
```

Complete the following steps:

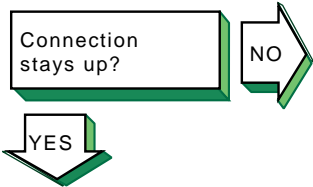
1. Verify that the addresses on the point-to-point link to the switch are correct by using the `ifconfig ipsx` command.
2. Verify that there is network connectivity to the IP controller on the switch by using the `ping` command. If the command fails, the local host's `ifconfig` command parameters might be incorrect. On the switch, possible reasons include: the IP controller is down or has an interface problem. Contact the switch administrator.
3. Verify that there is an `ips` route to the remote host's subnet by using the `netstat -r` command.



If the `ping` command fails, complete the following steps:

1. Check the cabling between the local host and the switch.
2. Verify that the default SNAP virtual circuit (VC) specified on the local host matches the default SNAP VC on the switch.
3. Contact the remote system administrator and verify that the remote system is up and running and that it is configured for IP switching correctly.
4. Verify the route to the remote host by using the `tracert` command. If the first hop in the output shows the default network interface and not the IP controller, add a static route to the remote subnet through the IP controller to your routing table. Issue the `netstat -r` command to verify the change.

If the route reaches the IP controller but goes no further, contact the remote system administrator to verify that the system is configured correctly and that the routing tables are correct.



Problem still exists?
Report your problem to
DIGITAL. See the *Network
Administration* guide.

If the connection terminates abnormally, complete the following steps:

1. Test the network to determine whether the problem is on the local host, remote host, or a host on the path between the two. See *Network Administration* for more information on testing the network.
2. Once you have identified the host with the problem, do the following:
 - a. Verify that the network mask for the local host is properly set up in the `/etc/rc.config` file.
 - b. Confirm that the network device is properly configured.
 - c. Make sure the local host's `hosts` database has the correct IP addresses.
 - d. Make sure the cabling from the local host to the network is intact and properly connected.

ATM Subsystem General Features

This chapter describes the following ATM features that writers of ATM device drivers, signaling modules, and protocol convergence modules can use:

- Header files
- Kernel data file
- Module configuration management
- Error codes
- Data formats
- Physical point of attachment
- Memory allocation
- ATM locking macros
- Types of circuits
- Global data structures

3.1 Header Files

When building kernel modules to interface with the DIGITAL UNIX ATM subsystem, your application must include the following header files (located in the `/usr/include/io/atm/sys` directory):

- `atm.h` — Common ATM subsystem structures, function prototypes, and ATM error codes (required by all ATM kernel modules)
- `atm_adi.h` — Device driver interface function prototypes and structures (required by all ATM device drivers)
- `atm_smi.h` — Signaling protocol module interface function prototypes and structure definitions (required by all signaling protocol modules)
- `atm_cmi.h` — Protocol convergence module interface function prototypes and structure definitions (required by all protocol convergence modules)
- `atm_osf.h` — Macros and constants for DIGITAL UNIX services that ATM subsystem modules use

In addition, the protocol convergence module code must include any signaling protocol-specific module header files that define structures used by protocol convergence modules.

The `atm.h` header file defines all the global data structures as well as the revision level of these structures. The `ATM_REVISION` constant, which is defined in this file, is passed to the CMM by device drivers, signaling modules, and convergence modules during their registration process. The CMM uses this constant to determine which version of global structures the modules use; it should not be changed.

3.2 Kernel Data File

The kernel data file (`/sys/data/atm_data.c`) is supported in DIGITAL UNIX Version 4.0 for backward compatibility only and will not be supported in future releases. Therefore, DIGITAL recommends that all ATM modules use the standard kernel module configuration manager to configure module initialization routines and module tunable parameters.

3.3 ATM Module Configuration

All ATM modules should use the standard DIGITAL UNIX configuration management interface for initializing and configuring modules. This interface also provides the `sysconfig` utility, which allows you to tune ATM modules without having to use a data file or recompile code. See *Writing Device Drivers: Tutorial* and `sysconfig(8)` for information on the DIGITAL UNIX configuration management interface and `sysconfig` utility, respectively. See Chapter 11 and *System Configuration and Tuning* for information on tuning the ATM subsystem.

3.4 Error Codes

The `/usr/sys/include/io/atm/sys/atm.h` header file contains ATM-specific error codes that are returned by all modules within the ATM subsystem to indicate the result of an operation. These codes consist of standard User-Network Interface (UNI) 3.0/3.1 error numbers and error numbers specific to DIGITAL UNIX.

When a function call is completed with no errors, the function must return the value `ATM_CAUSE_GOOD`. Unsuccessful completion is indicated by the return of some other value. Only the values listed in the `atm.h` header file can be returned. Routines returning error codes to system calls or management routines must return standard DIGITAL UNIX error codes as defined in the `sys/errno.h` file.

All ATM subsystem routines use type `atm_error_t` to indicate the outcome of an operation and all ATM subsystem modules must also do so. An `atm_error_t` can take on one of the values listed in `atm.h`. The ATM signaling modules are responsible for translating values to external network representations when necessary.

3.5 Data Formats

The type of data transmitted and received on a virtual circuit (VC) depends on the type of ATM Adaption Layer (AAL) protocol the convergence module specifies when setting up the VC.

The type of data a convergence module and the CMM exchange depends on the Segmentation and Reassembly (SAR) capabilities of the underlying device driver and hardware adapter. Convergence modules must be prepared either to handle drivers of differing capabilities or to recognize a driver that does not support the capabilities required by the convergence module. The convergence module interfaces provide a mechanism for convergence modules to obtain a driver's capabilities for making this determination.

Some device drivers or adapters implement some SAR functions in hardware or in a combination of hardware and software. This makes sending and receiving specific types of data more efficient. For example, TURBOchannel and Peripheral Component Interconnect (PCI) ATM adapters implement AAL5 in hardware. Other hardware may implement AAL3/4 or may be able to handle only raw ATM cells. Since the DIGITAL UNIX ATM subsystem is designed to accommodate a wide variety of drivers, it provides mechanisms for dealing with drivers of varying capabilities. It is the convergence module's responsibility to use the capabilities of a driver or an ATM interface in the best way possible for the protocol it is implementing.

The DIGITAL UNIX ATM subsystem can handle the following types of ATM data:

- Raw ATM cells (full ATM cells, including cell headers)
- Cooked packet Protocol Data Units (PDUs)

3.5.1 Raw ATM Cells

If a convergence module needs to transmit raw ATM cells, it should first determine if the underlying driver supports this service (some drivers may support only cooked data). If the driver cannot handle raw cells, the convergence module should not make or accept connections over the

interface controlled by the driver. The convergence module should use the driver only if the driver provides compatible capabilities.

3.5.2 Cooked Data

If a convergence module needs to transmit cooked data (data packets, such as AAL5, rather than individual cells), it should first determine if the underlying driver supports this service (some drivers may support only raw data or may not cook the data in the needed AAL format). The convergence module should use the driver if the driver provides compatible capabilities. If a driver does not support the required capabilities, but does support raw cells, the convergence module can perform the SAR functions internally and then send and receive raw cells. Otherwise, the convergence module should not accept or make connections over the interface controlled by the driver.

3.5.3 How the Data Is Carried

Both cell and packet data in the ATM subsystem are carried in a chain of mbufs. The routines for allocating and manipulating the mbufs are the same as those the DIGITAL UNIX networking subsystem uses. The data representation within the mbuf chain depends on the type of data (raw or cooked) being carried by the mbufs. When an mbuf chain contains raw cells, each mbuf in the chain contains exactly one ATM cell, a full 53-byte cell with Generic Flow Control (GFC) and Payload Type Indicator (PTI) bits set.

An mbuf chain can contain one or more ATM cells, each in its own mbuf. When an mbuf chain is carrying packet Protocol Data Units (PDUs) (AAL3/4 or AAL5 PDUs), the mbuf chain will contain exactly one packet PDU. On outgoing packets, the number of bytes in the packet must be less than or equal to the maximum transmission unit (MTU) of the VC on which the packet is transmitted. The mbuf chain must contain the appropriate headers and trailers (the AAL3/4 or AAL5 Convergence Sublayer Protocol Data Unit (CS-PDU) headers and trailers). If the device driver or adapter adds headers and trailers, the driver must strip off this extra information on outgoing packets. The drivers must include the CS-PDU headers and trailers on received packets.

3.5.4 Time-Stamping

When a driver receives raw ATM cells, the driver must be capable of optionally time-stamping the incoming cells to assist convergence modules or other protocol modules in determining intercell intervals; this might be required when processing constant bit rate (CBR) or variable bit rate (VBR) traffic. If a convergence module enables time-stamping of incoming data,

the device driver must add an 8-byte time-stamp immediately after the cell bytes in the mbuf, increasing the mbuf length to 61 bytes.

On DIGITAL UNIX systems, the time-stamp is the value read from the free-running system clock at the time the packet was received by the driver. This value has a 10-nanosecond resolution, but the value is accurate to only a few microseconds. This is because the time-stamp is not generated by the hardware when the packet arrives, but by the driver during its receive interrupt processing.

The driver can also optionally indicate lost or corrupted cells in the data stream. Cell loss can be due to hardware errors, the failure of cyclical redundancy checks (CRC), or insufficient resources to hold the incoming cell. The actual cause of local cell loss can vary between drivers. To indicate a lost cell in the data stream, the driver should insert a 0 length mbuf in the data stream where the cell was lost. Only one of these needs to be inserted if multiple sequential cells are lost. If time-stamping is enabled, the lost cell indicator will be an 8-byte mbuf with the time-stamp indicating the temporal position of the lost cell. The protocol convergence module can use this information for clock recovery, if necessary.

3.6 Physical Point of Attachment

The physical point of attachment (PPA) is a network services endpoint. It is the point at which the network services are provided and to which network services users attach to the network services. Each PPA represents a unique addressable entity on the network, identifying a provider of network services on both the local system and the network. When a call is placed on an ATM network, only the PPA is addressed; the PPA's address is registered with the ATM switch.

Although the network can address the PPA through the PPA address, the network cannot use the same address to address individual network services users attached to the PPA. For this, the ATM network uses ATM End System Addresses (AESAs). An AESA uniquely identifies both the network services endpoint and a specific network services user attached to that endpoint.

UNI 3.0/3.1 AESA-format ATM addresses consist of the following parts:

- A prefix — Assigned by the switch
- End system identifier (ESI) — Assigned by the local host
- A selector byte — Used by the network services provider on the end system to route a call to a specific network services user attached to a PPA

The first two parts together are registered with the ATM switch and network, and uniquely identify a PPA to which network services users may attach. Both the host and the switch must combine these two pieces of information to form complete addresses. This is done through the Integrated Local Management Interface (ILMI) protocol when new addresses are added or deleted. The CMM, in cooperation with signaling modules, keeps track of these two parts so that all addresses associated with either a prefix or an ESI are properly managed. Since the prefix is assigned by the switch, the CMM leaves the management of the prefixes up to the address registration and management portion of the signaling module.

ESI values have the following sources:

- The ESIs configured in the ATM adapter's read-only memory (ROM)
- ESIs configured by the system or network administrator, who uses the `atmconfig` command
- ESIs configured by convergence modules that need to create their own ATM addresses

The first two ESIs are considered global or public ESIs. All PPAs created from global ESIs are available to all convergence modules. Only the system or network administrator can delete global ESIs. ESIs that convergence modules create are considered private. All PPAs created from private ESIs are available only to the convergence module that configured the ESI on the system. These ESIs can be deleted by either the system or network administrator or by the creating convergence module. The CMM keeps track of ESI sources (in the `atm_esi` structure) to assure that only the correct entity can remove an ESI (and all associated PPAs and the VCs associated with those PPAs).

Since ATM permits an arbitrary number of ESIs and prefixes to be registered for a specific connection to the network, an end system can have an arbitrary number of PPAs. Users of the ATM network services (convergence modules) can attach their service to any combination of PPAs to make their service available on the network. The process of attaching a service to a PPA is called binding, and creates a full AESA that uniquely identifies a specific instance of a service on the network. In the DIGITAL UNIX ATM architecture, all calls must be placed and received through bind points that represent valid AESAs.

In the DIGITAL UNIX ATM subsystem, PPAs are created whenever an address is successfully registered with the local network switch. When a PPA comes into existence, convergence modules are notified and may bind their services to the PPAs. Each PPA contains three pieces of information that uniquely identify it among the different networks to which the host

may be attached. Associated with each PPA is a device driver (representing the physical connection to a network), a signaling module (representing the protocol that controls the administration of the PPA), and the PPA's address on the network to which it is attached. This permits PPAs on separate, disjoint networks to have the same logical address, but to still be unique on the local system.

The DIGITAL UNIX ATM subsystem has two types of PPAs:

- Permanent virtual circuit (PVC) PPAs, which manage PVCs
- Switched virtual circuit (SVC) PPAs, which manage SVCs

You can specify as many services for PVC connections as for SVC connections.

3.6.1 Permanent Virtual Circuit PPAs

When an interface becomes active on a network, the CMM automatically creates a special PPA for use in connecting PVCs to network services users. A PVC PPA is a special PPA that has no network address and no associated signaling module. It represents a path between the network and a network services user. There is exactly one PVC PPA for each interface that is operational. PVC PPAs are destroyed only when a driver is taken down or unregistered with the CMM. PVC PPAs remain in place throughout network disruptions.

Since PVC PPAs have no associated address, you must decide how to bind a specific network services user to a PPA and how to uniquely identify the user on that PPA. Since PVCs are created locally either with the `atmconfig` administration command or by convergence modules, and do not have any identifying characteristic that is unique on the network, you can assign an arbitrary addressing scheme to PPA bindings. The only requirements are that each network services user be uniquely identifiable and that there be a large enough address space to accommodate as many services as all the SVC PPAs can bind. See `atmconfig(8)` for more information.

In the DIGITAL UNIX ATM subsystem, a PVC bind point is identified not by an AESA, but by values unique to each network services user (convergence module) that binds to a PVC PPA. These values are the convergence module's name (provided when the convergence module registers with the CMM) and a selector value used to specify a specific instance of a service on the convergence module. Thus, a PVC network services user is identified by a driver/convergence/selector tuple that identifies a specific instance of a specific service on a specific network. The selector space for PVC bind points is 31-bits wide and is local to each convergence module; each convergence module has complete control over

the selector values used to access instances of its service. This provides a large enough bind space to enable convergence modules to create a PVC binding for every SVC binding and to accommodate both PVC and SVC access to its services.

3.6.2 Switched Virtual Circuit PPAs

When an ATM network interface is brought up, the signaling protocol that uses the interface must contact the local switch to register the end system with the network. For UNI 3.0/3.1, this involves exchanging addressing information with the switch to create full ATM addresses, minus the selector values. Each ATM address comprises a prefix value, which is assigned by the network, and an ESI, which identifies the end system. ATM permits the switch to assign an arbitrary number of prefixes and the end system to provide an arbitrary number of ESIs, within reason and subject to implementation-specific limitation. Each combination of prefix and ESI forms an ATM address, or a network services endpoint (PPA).

When an address is registered with a switch, the registering signaling module informs the CMM that a new network services endpoint exists. The CMM then creates a PPA for the new address and notifies each convergence module or the network services users of the new PPA so that they can create AESA bindings, if necessary. Calls to remote systems are given the calling party number of the bind point and PPA through which the call is placed. Incoming calls are routed to bind points based on the called party's AESA address. Therefore, all SVC activity must use a bind point as the local endpoint object. Calls received for an unbound AESA are rejected, and no calls can be placed without first creating a bind point and calling party's AESA.

The bind space for SVC PPAs is 8 bits long, the size of an AESA selector. If a convergence module needs a larger binding space, it can create more PPAs to which it can bind additional instances of its services. A convergence module can create an SVC PPA for its private use by defining a new ESI to the system. By defining a new ESI, one or more new PPAs are created with the new ESI and the current prefixes provided by the switch. The DIGITAL UNIX ATM subsystem recognizes PPAs made with ESIs created by a convergence module and makes the new PPAs available only to the module that created the ESI. This allows convergence modules to create their own address space on the ATM network.

ESIs taken from an adapter's ROM and ESIs created by using the `atmconfig` command are considered global. All resulting PPAs are not restricted to specific convergence modules. All other ESIs and their resulting PPAs are considered private.

When a driver is taken down or a connection to the network is lost, the CMM destroys all SVC PPAs, along with any bind points and VCs associated with the PPAs. If the connection to the network is reestablished, the signaling module must reregister all the ESIs; only the creator of the ESI or the system administrator, using the `atmconfig` command, can remove ESIs. The PPAs are then recreated. The convergence modules must bind to the new PPAs.

3.7 Memory Allocation

All modules that allocate memory for use in the ATM subsystems should use the same allocation mechanism. This enables the system to keep track of memory allocated for use by ATM and provides a consistent programming model. The following macros for allocating and freeing memory can be safely called in any context:

```
ATM_MALLOC(pointer,cast,size)
ATM_MALLOCW(pointer,cast,size)
ATM_MALLOC_VAR(pointer,cast,size)
ATM_MALLOCW_VAR(pointer,cast,size)
ATM_FREE(pointer)
```

The `ATM_MALLOC` and `ATM_MALLOCW` macros allocate memory of at least `size` bytes. They place the address of the allocated memory into `pointer`, casting it with the type of `cast`. The allocated memory is correctly aligned for any operation. You should use these macros if the size of the allocation is determined at compile time, as they are optimized for that purpose.

The `ATM_MALLOC_VAR` and `ATM_MALLOCW_VAR` macros perform the same function as `ATM_MALLOC` and `ATM_MALLOCW`. You should use these macros when the allocation size is computed at run time.

The `ATM_MALLOC` and `ATM_MALLOC_VAR` macros do not block if the requested memory is not available, but return `NULL`. You should use these macros in the majority of cases since any routine called by the ATM Subsystem is not allowed to block (except Module Management Interface (MMI) calls).

The `ATM_MALLOCW` and `ATM_MALLOCW_VAR` macros block until the requested memory is available. However, they can still return `NULL` if an error occurs during the allocation processing. Use these macros only in contexts that can block (system calls and private kernel threads). All memory allocated by these macros is mapped in kernel virtual memory and can be passed to any kernel function.

The `ATM_FREE` macro frees memory allocated previously by any `ATM_MALLOC` macro. The value of `pointer` must be identical to that

returned from the `ATM_MALLOC` macro. Once memory is freed, do not reference it again.

The macros return `NULL` if no memory is available. All modules must check the return value before dereferencing them to assure that memory was successfully allocated. Modules should also be prepared to handle situations where memory is not available; this should not be a fatal error.

The following code fragment shows how to use these macros:

```
struct my_struct *msp;

ATM_MALLOC(msp,struct my_struct *,sizeof(struct my_struct));
if(msp == NULL)
{
    /* allocation failed */
    return ATM_CAUSE_NOMEM;
}
/* use memory referenced by msp */
ATM_FREE(msp);
```

3.8 ATM Locking Macros

The base ATM subsystem is symmetric multiprocessing (SMP) and realtime (RT) safe. The CMM is highly parallelized and supports fine-grain concurrency in all attached modules. ATM modules that are included with the base ATM subsystem are either parallelized or funnel to the master processor. Therefore, any user-created ATM module must be both SMP and RT safe. The module must at least use funneling to force the module to run on the master processor. The module can also use fine-grain concurrency and locking.

The following section provides a brief overview of locking as it applies to ATM modules. See *Writing Device Drivers: Advanced Topics* for a complete description of locking within the DIGITAL UNIX kernel.

3.8.1 Locking Guidelines

Because of the way in which the ATM subsystem operates, you should write all routines of any ATM module that is called by the CMM to execute on an interrupt stack. Many ATM functions execute either off a device interrupt, the system soft clock, or an internal ATM thread. This means that no ATM routine that the CMM calls can block (with the exception of the `xxx_mmi` routine described in Appendix A). Therefore, all locks must be nonblocking simple locks. You can use blocking locks only from within a module's private thread, or when in a system call context.

No ATM module should hold a simple lock when calling outside the module (in general, simple locks should be held the minimum amount of time). Thus, when calling any CMM routine or any other kernel routine, no locks should be held. When the CMM calls any module's routine, no CMM locks are held across the call.

Finally, before taking a simple lock, raise the processor priority to `splimp` so that interrupts are blocked if the locking thread happens to be running on the master processor.

3.8.2 Types of Locking Methods

Modules that do not implement high levels of parallelism can implement one of the following locking methods (in increasing order of preference):

- Coarse-grain locking
- Funneling
- Using threads

Coarse-grain locking uses one lock (or perhaps a few) to lock a large code path or an entire module. Avoid this type of locking because it requires that a simple lock be held for long periods of time, possibly causing all other processors to block while waiting on the lock. Also, if the lock is held through function calls outside the module, locking hierarchy violations or deadlocks could result.

Funneling forces the module's thread of execution onto a single CPU. Since the module runs only on the master, it is effectively running in a uniprocessor environment. Once funneled, all calls outside the module also run on the master processor even though the called routine may be parallelized. Funneling causes severe context switch overhead that adversely affects module, ATM, and system performance. This is because the module's thread is suspended on the slave and then placed in the master processor's scheduling queue.

To funnel a thread, place the `unix_master()` function call at the start of and the `unix_release()` function call at the end of the code block to be funneled. There must be a corresponding `unix_release()` call for every `unix_master()` call. You must account for this when designing error paths through a code block.

Creating a thread for the module and then queuing work to the thread is the preferred option. All the module's work is performed in the thread that can run on only one processor (any processor in a multiprocessor environment). Thus, within the thread's code path no locks are required. Only the queuing mechanism, which enqueues and dequeues requests to the thread, requires locking.

Using this strategy, all module routines that can be called from outside the module would allocate some queue structure, take the arguments to the routine along with some identifier to identify the routine that has been called, place them in this structure, and then enqueue the structure to a service queue (locking service queue access). Once the structure is enqueued, the module's thread would be scheduled through one of the thread-management calls. Then, when the thread is run, it would dequeue a request from the service queue (again, locking the service queue) and process the request in the thread's context. The thread would run until there were no more requests on the service queue. This does incur the overhead of a context switch on a uniprocessor system, but on a multiprocessor system the enqueueing could take place on one processor at the same time the thread is being run on another.

3.8.3 Order of Locking Macros

The ATM locking functions are really macros that invoke other macros, and are defined in the `atm_osf.h` file. The following sequence of steps lists the locking macros and the order in which you call them in your module:

1. Declare and initialize the lock information structure.

```
atm_lock_info(lock_info_name);
```

This declares the lock information structure and allocates storage for it. Lock information structures are declared as globals, so this macro should appear outside all functions. When creating a lock for structures that are dynamically allocated, you need only a single lock information structure that can be applied to all instances of the structure.

2. Declare a lock.

```
atm_lock_decl(lock_name)
```

Note

Do not put a semicolon at the end of the declaration.

This declares the lock and allocates storage for it. You can use the macro within a structure declaration or by itself in a global context.

3. Initialize the lock.

```
atm_lock_setup(&lock_name,&lock_info_name);
```

Be sure to pass the address of the lock and lock information structures in this macro.

4. Take the lock.

```
atm_lock(&lock_name);
```

Be sure to raise the processor priority immediately before taking the lock.

5. Release the lock.

```
atm_unlock(&lock_name);
```

Be sure to lower the processor priority immediately after releasing the lock. Do not lower the processor priority while the lock is held.

6. If the lock is no longer needed, (for example, when storage for a structure with an embedded lock is being released), terminate the lock.

```
atm_lock_terminate(&lock_name);
```

Once terminated, you can no longer use the lock unless you reinitialize it.

The following code fragment shows the sequence of calls in creating and using a lock:

```
atm_lock_decl(some_lock)
atm_lock_info(,some_lock_info);
int oldpri;

/* initialize the lock */
atm_lock_setup(&some_lock,&some_lock_info);

/* take the lock */
oldpri = splmp();
atm_lock(&some_lock);

/* release lock */
atm_unlock(&some_lock);
splx(oldpri);

/* terminate the lock when it will no longer be used */
atm_lock_terminate(&some_lock);
```

3.8.4 Creation of ATM Threads

ATM modules can create kernel threads for the following reasons:

- To schedule work in the module to be performed in a context that is private to that module. For example, a module may process incoming data from a thread rather than on the interrupt stack (incoming data is passed to convergence modules on the driver's interrupt stack).

- To perform functions that have to occur at periodic intervals such as protocol timeouts or garbage collection.

The DIGITAL UNIX kernel provides many kernel thread primitives that modules are free to call directly. These are described in greater detail in *Writing Device Drivers: Advanced Topics*. However, to create a thread for use by an ATM module, the ATM subsystem provides the `atm_cmm_new_thread` function. This function combines many of the DIGITAL UNIX thread primitives to provide an easier interface for ATM modules. See Appendix A for a description and format of this function.

3.9 Types of Circuits

The ATM subsystem supports the following circuits:

- Available bit rate (ABR)
- Constant bit rate (CBR)

A circuit in which both end systems and the network dedicate the resources necessary to handle transmission and reception of the circuit's traffic at the specified bit rate. CBR circuits requiring end-to-end timing are not supported because the drivers and adapters do not support the AAL1 capability.

- Pacing

A circuit that allows convergence modules to specify that the local driver perform cell pacing on non-CBR circuits. It requires that the sending system dedicate resources to insure that traffic is injected onto the network at the specified bit rate. A pacing circuit has local significance only. To the network and target system, the pacing circuit is a best-effort ABR connection. You can use pacing circuits, for example, to limit the local host's transmission rate without having both end systems and the network treat the circuit as CBR.

The default best-effort ABR connections do not require the end systems to dedicate resources (such as bandwidth) to the connections. However, CBR and pacing connections require the end systems to dedicate resources to those connections. These resources can either be allocated transparently at the time the connection is made or received, or they can be reserved in advance by the sending or receiving convergence module and later applied to a connection setup or incoming call. See Section 7.1 and Section 7.2 for information on reserving resources for connections.

3.10 Global Data Structures

The following data structures are visible to all modules of the ATM subsystem and to the protocol convergence module:

- `atm_vc`
- `atm_addr`
- `atm_vc_services`
- `atm_uni_call_ie`
- `atm_ppa`
- `atm_esi`
- `atm_cause_info`

The ATM subsystem uses these structures to keep track of information for each VC, such as VC service parameters, connection endpoint addresses, VC numbers, and traffic statistics. When a module other than the CMM must allocate memory for a structure, the CMM provides a function call or macro to allocate structure memory in a consistent manner, to properly initialize the structures, and to allocate the correct version of the structure. Module writers must not use any routines other than those that the CMM supplies to allocate structure memory. The following sections describe these data structures.

3.10.1 The `atm_vc` Structure

The ATM subsystem uses the `atm_vc` structure to reference a VC. A VC is the object on which data is sent and received. VCs are associated with one or more local endpoints and one or more remote endpoints. An action performed on a VC usually affects all the endpoints associated with the VC. For example, transmitting data on a VC sends the data to all endpoints currently connected to the VC.

The `atm_vc` structure has the following characteristics:

- Only the CMM allocates and frees the structure.
- The CMM and other ATM modules use the structure for keeping track of an active VC. The CMM also uses it to maintain all per-VC state information.
- Some structure members are reserved for use by the various modules in the ATM subsystem. This enables all ATM modules to use a common reference for a specific VC. All modules use the pointer to this structure as the handle for a specific VC.
- Each module of the ATM subsystem can write only those members of the structure that are assigned for its use. All other structure members should be considered read-only.
- The structure's size and members might change in the future, but the arrangement of the per-module structure members should not change.

- No locking is required for access to this data structure. Since each module is permitted to modify only those fields assigned to it, there is no need to coordinate access to the entire structure with other modules. Access within a module may be locked if necessary.
- You use a pointer of type `atm_vc_p` to reference the structure.

When a module is first informed of the presence of a new connection, it receives a reference to an `atm_addr` structure for the connection endpoint. The `atm_addr` structure references the `atm_vc` structure for the connection. At this point, the module can use its private structure members in the `atm_vc` structure to hold information necessary to keep track of the VC and can use its private pointer structure members to store any information it requires.

When the VC is torn down and local resources are deallocated, the CMM calls each module to deallocate any resources associated with the VC. At this time, the module must zero out its `atm_vc` structure private members to signify that it no longer has any reference to the defunct VC. The CMM checks all the `atm_vc` private members before freeing the structure memory and prints a warning to the console if any private structure member is not 0.

Table 3–1 lists those member names of the `atm_vc` structure, with their associated data types, that modules might reference.

Table 3–1: The `atm_vc` Structure Members

Member Name	Data Type
<code>conv_pp1</code>	<code>void *</code>
<code>conv_pp2</code>	<code>void *</code>
<code>sig_pp1</code>	<code>void *</code>
<code>sig_pp2</code>	<code>void *</code>
<code>drv_pp1</code>	<code>void *</code>
<code>drv_pp2</code>	<code>void *</code>
<code>vcs</code>	<code>atm_vc_services_p</code>
<code>call_reference</code>	<code>long</code>
<code>errno</code>	<code>atm_error_t</code>
<code>vci</code>	<code>int</code>
<code>vpi</code>	<code>int</code>
<code>ppa</code>	<code>atm_ppa_p</code>

Table 3–1: The atm_vc Structure Members (cont.)

Member Name	Data Type
selector	int
direction	atm_direction_t

3.10.1.1 The conv_pp1 and conv_pp2 Members

The `conv_pp1` and `conv_pp2` members are pointers reserved for the use by convergence modules only; no other ATM module is permitted to use these structure members. Typically, a convergence module uses these members to reference local structures and resources associated with the connection that the `atm_vc` structure controls. However, convergence modules may use these fields in other ways.

3.10.1.2 The sig_pp1, sig_pp2, drv_pp1, and drv_pp2 Members

The `sig_pp1`, `sig_pp2`, `drv_pp1`, and `drv_pp2` members have corresponding meanings for ATM signaling protocol modules and ATM device drivers, respectively.

3.10.1.3 The vcs Member

The `vcs` member is a pointer to an `atm_vc_services` structure that contains all the service parameters for the VC. The CMM sets this pointer.

Note

This member and any member of the structure that it references should never be written by any module other than the CMM.

3.10.1.4 The call_reference Member

The `call_reference` member is a unique call identifier assigned to the VC. This value is assigned only by the CMM and signaling protocol modules and must not be modified by any other module.

3.10.1.5 The errno Member

The `errno` member contains the last error number that was reported on the VC.

3.10.1.6 The vci and vpi Members

The `vci` and `vpi` members contain the virtual circuit's virtual path identifier (VPI) and virtual channel identifier (VCI) values. These are assigned by the CMM in cooperation with the signaling protocol module that set up the connection.

3.10.1.7 The ppa Member

The `ppa` member indicates the PPA to which the VC belongs.

3.10.1.8 The selector Member

The `selector` member is the selector value assigned to the bind point to which the VC is attached. This value combined with the address information in the PPA form the full AESA address.

3.10.1.9 The direction Member

The `direction` member provides information about the direction of the call. The CMM sets this member to one of the following:

- `ATM_DIRECTION_PVC`, if the VC structure references a PVC
- `ATM_DIRECTION_CALLING`, if the VC structure references a call placed by the local system
- `ATM_DIRECTION_CALLED`, if the VC structure references a call received by the local system.

This structure member is set only by the CMM and must not be modified by any other module.

3.10.2 The atm_addr Structure

The ATM subsystem uses the `atm_addr` structure to reference a connection endpoint. One `atm_addr` structure exists for every connection to every endpoint. Multiple endpoints and multiple address structures can be associated with a single VC (for example, point-to-multipoint connections). A pointer to this structure is the handle that the ATM subsystem uses to reference a specific connection endpoint.

The `atm_addr` structure has the following characteristics:

- The structure stores all address- and state-related information for a connection endpoint. This structure contains only endpoint information, not information about the connection to the endpoint. The connection information is maintained in the `atm_vc` structure that is referenced in

the `atm_addr` structure. Except at structure initialization time, all address structures are associated with a VC.

- All function calls that perform an action related to a specific endpoint use a pointer to this structure.
- The structure maintains local address information since the local address is also a connection endpoint.

The structure may contain additional members for the internal use of the CMM. ATM modules should make no assumption about the actual length of the structure. Table 3–2 lists those members of the `atm_addr` structure, with their associated data types, that modules might reference.

Table 3–2: The `atm_addr` Structure Members

Member Name	Data Type
<code>vc</code>	<code>atm_vc_p</code>
<code>address[20]</code>	<code>unsigned char</code>
<code>ton</code>	<code>unsigned char</code>
<code>anpi</code>	<code>unsigned char</code>
<code>subaddress[20]</code>	<code>unsigned char</code>
<code>subaddress_type</code>	<code>unsigned char</code>
<code>eprtype</code>	<code>unsigned char</code>
<code>endpoint</code>	<code>unsigned short</code>
<code>state</code>	<code>unsigned char</code>
<code>atm_error</code>	<code>atm_error_t</code>
<code>setup</code>	<code>void *</code>
<code>connect</code>	<code>void *</code>
<code>location</code>	<code>unsigned char</code>
<code>cause</code>	<code>atm_err_t</code>
<code>diag_length</code>	<code>unsigned char</code>
<code>diagnostic[27]</code>	<code>unsigned char</code>
<code>endstate</code>	<code>unsigned char</code>
<code>conv_p1</code>	<code>void *</code>
<code>conv_p2</code>	<code>void *</code>

Table 3–2: The atm_addr Structure Members (cont.)

Member Name	Data Type
sig_p1	void *
sig_p2	void *

3.10.2.1 The vc Member

The `vc` member references the `atm_vc` structure of the VC to which this endpoint belongs.

3.10.2.2 The address Member

The `address` member is an array that contains the 20-byte ATM address of the endpoint.

3.10.2.3 The ton Member

The `ton` member is a value that specifies the endpoint's address type. These bits are identical to the type of number field in the Called Party Number information element (IE) (right justified).

3.10.2.4 The anpi Member

The `anpi` member is the address or numbering plan identification information for the address. These bits are identical to those in the corresponding field of the Called Party Number IE.

3.10.2.5 The subaddress Member

The `subaddress` member is an array that contains the subaddress of the endpoint. If the endpoint has no subaddress, the Authority and Format Indicator (AFI) byte of the subaddress array must be 0.

3.10.2.6 The subaddress_type Member

The `subaddress_type` member is a numeric value that specifies the type of subaddress that the endpoint uses. These bits are identical to those in the Type of subaddress field of the Called Party Subaddress IE (right justified).

3.10.2.7 The eprtype Member

The `eprtype` member is the endpoint reference type of the endpoint referenced by the structure. Signaling modules fill in this information, indicating the type of endpoint contained in the endpoint structure member.

3.10.2.8 The endpoint Member

The `endpoint` member is the endpoint reference value that the ATM subsystem uses to reference an endpoint in a point-to-multipoint connection. The CMM assigns this value when processing point-to-multipoint connection requests.

3.10.2.9 The state Member

The `state` member is the call state value or global interface state value field from the last Call State IE received for the connection. Only the signaling module fills in this member.

3.10.2.10 The `atm_error` Member

The `atm_error` member contains the error number for the last error reported on the endpoint.

3.10.2.11 The `setup` Member

The `setup` member is a pointer to a signaling protocol-specific structure that contains the parameters sent or received in the call setup phase of connection creation. These parameters indicate the call parameters set by the calling party of a connection. On incoming calls (the local host is the called party), the signaling protocol module sets this member. On outgoing calls (the local host is the calling party), the protocol convergence module that initiates the call sets this member.

The protocol convergence module and the signaling module must agree on the format of the object referenced by this member. The protocol convergence module can free storage for the objects referenced only when the associated connection is torn down.

3.10.2.12 The `connect` Member

The `connect` member is a pointer to a signaling protocol-specific structure that contains the parameters sent or received in the connection phase of connection creation. These parameters indicate the actual parameters the ATM subsystem uses to establish the service between the endpoints. These could differ from the setup parameters in cases where parameter negotiation took place. Both sets of parameters are maintained so that any differences between requested and negotiated parameters can be easily determined.

The protocol convergence module and the signaling module must agree on the format of the object referenced by this member. The protocol

convergence module can free storage for the objects referenced only when the associated connection is torn down.

3.10.2.13 The location, cause, diag_length, and diagnostic Members

The `location`, `cause`, `diag_length`, and `diagnostic` members are filled in by the CMM from the Cause IE received for the connection. Convergence modules can use this information to determine the reason for a call failure. The diagnostic array contains `diag_length` valid bytes taken from the Cause IE length field.

3.10.2.14 The endstate Member

Only the signaling module fills in the `endstate` member as part of the enquiry processing.

3.10.2.15 The conv_p1 and conv_p2 Members

The `conv_p1` and `conv_p2` members are reserved for use by convergence modules only; no other module may access these members. Typically, these members store state information associated with the endpoint.

3.10.2.16 The sig_p1 and sig_p2 Members

The `sig_p1` and `sig_p2` members are reserved for use by signaling modules only; no other module may access these members. Typically, these members store state information associated with the endpoint.

No locking is required for access to this data structure. Since each module is permitted to modify only those fields assigned to it, there is no need to coordinate access to the entire structure with other modules. Access within a module may be locked if necessary.

3.10.2.17 Allocating the atm_addr Structure

One `atm_addr` structure is allocated by a protocol convergence module for each endpoint it calls and by signaling modules for each endpoint that calls the local host. Convergence and signaling modules use the `atm_cmm_alloc_addr` function call to allocate memory for the structure. The `atm_cmm_alloc_addr` call returns a valid pointer, if memory was successfully allocated and initialized; a NULL, if an error occurred. This is the only valid means to allocate memory for the structure.

Usually, the CMM frees all ATM address structures associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one

allocation fails), it may be necessary for the allocating module to free memory it has allocated. In these cases, modules can call the `atm_cmm_free_addr` function call with the value returned from `atm_cmm_alloc_addr` to free memory. Once the CMM accepts a connection request (either incoming or outgoing), only the CMM can free storage for this structure. See Appendix A for a description of the two routines.

3.10.3 The `atm_vc_services` Structure

Every connection on the system is allocated some amount of network resources (a portion of the network bandwidth). When a new connection is requested, the requester tells the ATM subsystem the type of physical resources required by the connection. The CMM ensures those resources are available and allocates them to the connection.

The CMM uses the `atm_vc_services` structure to keep track of connection resources and to inform the device drivers of the resources that must be allocated for a connection. The `atm_vc_services` structure has the following characteristics:

- One structure must be allocated and the members set to valid values for each new connection by the entity that creates the connection (convergence protocol module on outgoing calls or signaling protocol module on incoming calls). Once the connection service parameters are set you cannot change them.
- The CMM keeps track of these structures on a per-VC basis and frees them when the connection is torn down.
- The structure defines those services the CMM and driver manage. Every signaling and convergence module must properly set up the structure regardless of the signaling protocol used to actually set up the connection.
- Signaling protocol modules may use the structure to obtain information needed to create various signaling protocol messages used in call management (available to all modules).
- The structure is also used to track resource reservations. A convergence module can reserve resources for allocation to incoming or outgoing calls at a later time. You use this structure to define what resources are to be reserved and to track the reserved resources. See Section 7.1 for more complete information on the use of this structure in resource reservation.
- This structure defines parameters relative to the local system, independent of the actual call direction. The term *forward* means the sending side of the connection on the local system; the term *backward* means the receiving side of the connection on the local system. This is

different from the ATM Forum UNI specification, which defines parameters based on the call direction.

The signaling module must perform any conversions between the `atm_vc_services` structure members and the appropriate information elements (IEs) in the signaling message. This might require interchanging forward and backward fields, depending on the direction of the call.

- The structure is always referenced as a pointer of type `atm_vc_services_p`.

Table 3–3 lists those member names of the `atm_vc_services` structure, with their associated data types, that modules might reference.

Table 3–3: The `atm_vc_services` Structure Members

Member Name	Data Type
<code>vc</code>	<code>atm_vc_p</code>
<code>fqos</code>	<code>atmqos_t</code>
<code>bqos</code>	<code>atmqos_t</code>
<code>fmtu</code>	unsigned int
<code>bmtu</code>	unsigned int
<code>valid_rates</code>	unsigned int
<code>fpeakcr[2]</code>	unsigned int
<code>bpeakcr[2]</code>	unsigned int
<code>fsustcr[2]</code>	unsigned int
<code>bsustcr[2]</code>	unsigned int
<code>fburstcr[2]</code>	unsigned int
<code>bburstcr[2]</code>	unsigned int
<code>flags</code>	unsigned int
<code>aal</code>	<code>atmaal_t</code>
<code>queue</code>	unsigned int
<code>bearer_class</code>	unsigned int
<code>lerrstat</code>	enum <code>atm_lerr_t</code>
<code>nerrstat</code>	unsigned int
<code>cmm_drv_handle</code>	<code>atm_drv_handle_t</code>

Table 3–3: The atm_vc_services Structure Members (cont.)

Member Name	Data Type
drv_resource	void *
converge_handle	void *

3.10.3.1 The vc Member

The `vc` member is a pointer to the VC structure for the connection.

3.10.3.2 The fqos and bqos Members

The `fqos` and `bqos` members are the forward and backward quality of services (QOS) types, respectively. These indicate the QOS the caller is requesting for the connection. The following table lists the values and meanings for `fqos` and `bqos`:

Value	Meaning
ATM_QOS_CLASSA	Connection-oriented, constant bit rate (CBR) traffic with source or destination timing relationships.
ATM_QOS_CLASSB	Connection-oriented, variable bit rate (VBR) traffic with source or destination timing relationships.
ATM_QOS_CLASSC	Connection-oriented, variable bit rate (VBR) traffic with no timing relationships.
ATM_QOS_CLASSD	Connectionless, variable bit rate (VBR) traffic with no timing relationships.
ATM_QOS_CLASSX	Undefined bit rate traffic.
ATM_QOS_CLASSY	Available bit rate (ABR) traffic.
ATM_QOS_NONE	No specified quality of service.

3.10.3.3 The fmtu and bmtu Members

The `fmtu` and `bmtu` members are the forward and backward maximum transmission unit (MTU) size (in bytes), respectively. These indicate the maximum packet size that can be transmitted and received on the connection when the connection carries AAL3/4 or AAL5 data.

Only device driver modules use the `fmtu` and `bmtu` members to allocate resources for a connection; the signaling module does not use them to store MTU-related information elements (IEs). If a protocol requires explicit signaling of MTU information, the convergence module must use the `atm_uni_call_ie` structure to supply the IEs. See Section 3.10.4 for more

information about the structure and its members. See Chapter 7 for information on making outgoing connections and accepting connections.

3.10.3.4 The `valid_rates` Member

The `valid_rates` member is a bit mask that indicates which line rates have been specified and which have been left unspecified. The following table lists the values and meanings for `valid_rates`:

Value	Meaning
<code>ATM_VCRV_FPEAK0</code>	Forward peak cell rate for CLP=0
<code>ATM_VCRV_FPEAK1</code>	Forward peak cell rate for CLP=1
<code>ATM_VCRV_BPEAK0</code>	Backward peak cell rate for CLP=0
<code>ATM_VCRV_BPEAK1</code>	Backward peak cell rate for CLP=1
<code>ATM_VCRV_FSUST0</code>	Forward sustainable cell rate for CLP=0
<code>ATM_VCRV_FSUST1</code>	Forward sustainable cell rate for CLP=1
<code>ATM_VCRV_BSUST0</code>	Backward sustainable cell rate for CLP=0
<code>ATM_VCRV_BSUST1</code>	Backward sustainable cell rate for CLP=1
<code>ATM_VCRV_FBURST0</code>	Forward burst cell rate for CLP=0
<code>ATM_VCRV_FBURST1</code>	Forward burst cell rate for CLP=1
<code>ATM_VCRV_BBURST0</code>	Backward burst cell rate for CLP=0
<code>ATM_VCRV_BBURST1</code>	Backward burst cell rate for CLP=1

3.10.3.5 The `fpeakcr`, `bpeakcr`, `fsustcr`, `bsustcr`, `fburstcr`, and `bburstcr` Members

The `fpeakcr`, `bpeakcr`, `fsustcr`, `bsustcr`, `fburstcr`, and `bburstcr` members are the arrays that contain the forward and backward peak, forward and backward sustainable, and forward and backward burst cell rates, respectively. These indicate the network bandwidth requested by the caller. The first element of each array specifies the value for cells that have the Cell Loss Priority (CLP) bit clear (CLP=0). The second element of each array specifies the value for cells that have the CLP bit set (CLP=1). Only certain combinations of these rates may be valid for a given signaling protocol or for a specific QOS type. Those rates that are explicitly specified are designated by setting the appropriate bit in the `valid_rates` member.

Note

To set a cell rate of 0, set the appropriate cell rate and mark it as valid; do not leave the cell rate unspecified.

3.10.3.6 The flags Member

The `flags` member contains a bit that specifies additional services required for the connection. The following table lists the values and meanings for `flags`:

Value	Meaning
ATM_SERVICES_BEI	Indicates that the connection should use the best effort service.
ATM_SERVICES_BTAG	Indicates that the network should enable tagging in the backward direction.
ATM_SERVICES_CBR	Indicates that the local ATM interface should pace cells out to the network and that the circuit should be set up so that the requested bandwidth is guaranteed for the VC. This flag causes both local cell pacing to be enabled and the circuit to be signaled as a CBR circuit. CBR can be enabled only on hardware that supports this feature.
ATM_SERVICES_CLIPPING	Indicates that the network should enable clipping of cells on this circuit.
ATM_SERVICES_FTAG	Indicates that the network should enable tagging in the forward direction.
ATM_SERVICES_NOTIMING	Indicates that the circuit has no end-to-end timing requirements. If neither this flag nor the <code>ATM_SERVICES_TIMING</code> flag are set, the circuit is signaled as having unspecified timing requirements.
ATM_SERVICES_PACING	Indicates that the local ATM interface should pace cells out to the network as if the circuit were a CBR circuit. This flag has local significance only and does not affect any of the signaling messages used in creating the connection. This flag is used mainly when a convergence module needs to limit the rate at which cells can be transmitting on a specific VC. Pacing can be enabled only on hardware that supports this feature.

Value	Meaning
ATM_SERVICES_TIMING	Indicates that this circuit has some end-to-end timing requirements. When set, the circuit is signaled as having timing requirements. Also, the local system may enable features to facilitate the delivery of data with timing constraints.
ATM_SERVICES_VBR	Indicates that the local ATM interface and the network should treat this circuit as a variable bit rate circuit. This flag both enables VBR cell processing on the local adapter and causes the circuit to be signaled as a VBR circuit. VBR can be enabled only on hardware that supports this feature.

3.10.3.7 The aal Member

The `aal` member specifies the AAL type for the connection and can have one of the following values: `ATM_AAL1`, `ATM_AAL2`, `ATM_AAL3/4`, or `ATM_AAL5`.

3.10.3.8 The queue Member

The `queue` member specifies the driver send queue to use for the VC (for drivers that support multiple send queues). The CMM sets this value; it must not be set by any other module.

3.10.3.9 The bearer_class Member

The `bearer_class` member specifies the bearer class information to be used when signaling the call or that was signaled by the caller. For UNI signaling, this must be set to one of the following values: `ATM_BBEARER_BCOB_A`, `ATM_BBEARER_BCOB_C`, or `ATM_BBEARER_BCOB_X`.

3.10.3.10 The lerrstat Member

The `lerrstat` member contains error indications that result from the CMM checking the validity of the structure contents. The following table lists the values and meanings for `lerrstat`:

Value	Meaning
ATM_LERR_GOOD	Indicates that no errors were found in the structure contents.
ATM_LERR_PEAKMISSING	Indicates that the peak cell rate for CLP0+1 has not been specified. A peak cell rate must be specified on all circuits (though the rates may be specified as 0).
ATM_LERR_PEAKTOOBIG	The specified CLP0+1 rate exceeds the rate that is permitted by the hardware or the system configuration.
ATM_LERR_NOBWAFAIL	The specified peak CLP0+1 bandwidth exceeds the available bandwidth on the interface.
ATM_LERR_BADPEAKCR0	The specified CLP0 peak cell rate exceeds the specified CLP0+1 peak cell rate. The CLP0 cell rate must be less than or equal to the CLP0+1 cell rate, depending on the connection type.
ATM_LERR_BADBURSTCR0	The CLP0 burst cell rate exceeds the CLP0+1 burst cell rate.
ATM_LERR_BURSTTOOBIG	The specified CLP0 or CLP0+1 burst cell rate exceeds the specified peak CLP0+1 cell rate.
ATM_LERR_BADSUSTCR0	The specified CLP0 sustained cell rate exceeds the specified CLP0+1 sustained cell rate.
ATM_LERR_SUSTTOOBIG	The specified CLP0 or CLP0+1 sustained cell rate exceeds the specified CLP0+1 peak cell rate.
ATM_LERR_RATETOOSMALL	The specified peak cell rate is smaller than the minimum rate the local hardware can allocate. A peak cell rate of 0 is always allowed.
ATM_LERR_BADTRAFFTYPE	Invalid traffic flags specified. This error occurs when neither the CBR or PACING flags are set, but the circuit is being configured as a CBR circuit.
ATM_LERR_BADDRVHANDLE	When applying a reserved resource, the driver on which the resources were reserved does not match the driver on which the VC is being created.
ATM_LERR_RESNOTALLOC	When applying a reserved resource, the <code>atm_vc_services</code> structure is not backed by allocated resources. This indicates that an <code>atm_vc_services</code> structure that does not reference previously reserved resources is being applied in a context where reserved resources are required.

Value	Meaning
ATM_LERR_RESALREADY	When reserving resources, the <code>atm_vc_services</code> structure already references reserved resources. This indicates that an <code>atm_vc_services</code> structure that has already been used to reserve resources is being reused to reserve the same or different resources.
ATM_LERR_RATESCHANGED	When applying reserved resources, the information specified in the <code>atm_vc_services</code> structure has been changed since the resources referenced by the structure were originally reserved. Once resources are reserved, convergence modules are not allowed to change any of the entries in the <code>atm_vc_services</code> structure.
ATM_LERR_RATESINCOMPAT	When applying a reserved resource, the rates specified in the <code>atm_vc_services</code> structure are incompatible with the rates requested in an incoming call.
ATM_LERR_VCALREADY	When applying a reserved resource, the VC to which the resource is being applied already has a previously applied resource. Resources can be applied only once, and each VC can be assigned resources only once.

3.10.3.11 The `nerrstat` Member

Signaling modules use the `nerrstat` member to report signaling protocol-specific validity checking errors on the contents of the `atm_vc_services` structure. The values of this member depend on the signaling module. This structure is currently always set to zero (0).

3.10.3.12 The `cmm_drv_handle` Member

The `cmm_drv_handle` member contains the driver handle of the driver for which resources have been reserved (CBR and pacing circuits only). This may be compared to the driver handle information in the `atm_ppa` structure if a convergence module needs to determine whether reserved resources can be applied to a specific VC. This member is set by the CMM and must not be modified by any other module.

3.10.3.13 The `drv_resource` Member

Device drivers use the `drv_resource` member to store private information necessary to track reserved resources. The contents of this member have meaning only to the specific device driver on which resources have been

reserved, and are not examined by the CMM. Only the driver on which the resources have been reserved can modify this member.

3.10.3.14 The `converge_handle` Member

Convergence modules use the `converge_handle` member to help track reserved resources. The contents of this member have meaning only to the specific convergence module that requested the reservations of resources, and are not examined by the CMM. Only the convergence module that reserved resources can modify this member.

3.10.3.15 Allocating the `atm_vc_services` Structure

To allocate the `atm_vc_services` structure, use the `atm_cmm_alloc_services` function call. This call allocates memory for the structure and initializes its members to default values for best-effort available bit rate (ABR) service. Once allocated, other ATM modules can change the members of this structure as needed.

Typically, the CMM frees the `atm_vc_services` structure associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails), the allocating module might need to free memory it has allocated. In these cases, modules call the `atm_cmm_free_services` routine with the value returned from `atm_cmm_alloc_services` to free memory.

See Appendix A for syntax and a description of the two routines.

3.10.4 The `atm_uni_call_ie` Structure

The CMM uses the `atm_uni_call_ie` structure to store optional information elements (IEs) when the UNI 3.0/3.1 signaling protocol module sets up a VC. Since UNI 3.0/3.1 is the default signaling protocol that the DIGITAL UNIX ATM subsystem uses, the structure is described in this section. The `atm_uni_call_ie` structure has the following characteristics:

- The structure communicates only connection setup information between convergence protocol modules and the UNI 3.0/3.1 signaling protocol module.
- The CMM does not access or modify this structure.
- The information specified in this structure is taken directly from the various optional IEs used in the SETUP and CONNECT signaling messages. Mandatory IEs necessary for connection setup are built by the UNI 3.0/3.1 signaling protocol from information taken from the

atm_addr and the atm_vc_services structures. See Section 3.10.2 and Section 3.10.3, respectively, for more information.

- The structure contains a union of all IEs that a convergence module can specify.
- Each instance of this structure specifies exactly one IE. To specify multiple IEs, you must define multiple instances of this structure. Convergence modules can specify as many IEs as are needed to place the call.
- The structure defines the terms forward and backward using the ATM Forum UNI specification definitions. Forward is the direction from the caller to the callee and backward is the direction from the callee to the caller. This is in contrast to the definitions used in the atm_vc_services structure. See Section 3.10.3 for more information.

Example 3–1 shows the atm_uni_call_ie structure definition.

Example 3–1: The atm_uni_call_ie Structure Definition

```
struct atm_uni_call_ie {
    atm_ie_types_t  ie_type;
    char            last;
    union {
        union {
            struct {
                unsigned short  subtype;
                unsigned short  cbr_rate;
                unsigned short  multiplier;
                unsigned short  crt;
                unsigned short  err_corr;
                unsigned short  sdt;
                unsigned short  pfc;
            } aal1;
            struct {
                unsigned int     fsdu;
                unsigned int     bsdu;
                unsigned int     mid;
                unsigned short   mode;
                unsigned short   sscs;
            } aal34;
            struct {
                unsigned int     fsdu;
                unsigned int     bsdu;
                unsigned short   mode;
                unsigned short   sscs;
            } aal5;
            unsigned short user_aal[4];
        } aal_params;
    }
};
```

Example 3–1: The atm_uni_call_ie Structure Definition (cont.)

```
struct {
    unsigned short  hlit;
    unsigned short  hlistz;
    unsigned short  hli[8];
} bb_high_layer;
struct {
    unsigned short  layer2proto;
    unsigned short  mode2;
    unsigned short  q933;
    unsigned short  window_size;
    unsigned short  user2proto;
    unsigned short  layer3proto;
    unsigned short  mode3;
    unsigned short  dpsize;
    unsigned short  pktwindow;
    unsigned short  user3proto;
    unsigned int    ipi;
    unsigned short  snapid;
    unsigned short  oui[3];
    unsigned short  pid[2];
} bb_low_layer;
struct {
    unsigned short  bbri;
} bbrepeat;
unsigned short  bytes[28];
} ie;
};
```

3.10.4.1 The ie_type Member

The `ie_type` member indicates the type of IE specified in the current instance of the structure. You must specify exactly one IE type per structure instance.

3.10.4.2 The last Member

The `last` member indicates the last element of the IE array. This member must have a nonzero value for the last element of the array and a value of 0 for all other elements. The last element of the array does not need to be the last physical element of the array. This indicates the last logical element so that a module reading the array can determine where the last valid element is located. To declare an element between the start of the array and the last element as unused, set the element's type to `ATM_IET_NONE`.

3.10.4.3 The aal_params Member

The `aal_params` member is a union that contains the data to be placed in (or read from the SETUP message) the AAL Parameters IE in the SETUP and CONNECT signaling messages. The way the system interprets this union depends on the value of the `ie_type` member. The meaning of each member of each structure in the `aal_params` union corresponds to a data field in the AAL Parameters IE. See the UNI 3.0/3.1 specification for details on the meaning and use of these IE fields.

3.10.4.4 The bb_high_layer and bb_low_layer Members

The `bb_high_layer` and `bb_low_layer` members are structures that specify information from their corresponding IE. Data in these structures is stored in machine native form, not just as a series of bytes from the network. Information in each structure is set either as numeric values or from constants as defined in the `atm/sys/atm.h` file. Also, the IE headers (the first 5 bytes of each IE that identifies the IE type and length) are not stored.

Note

Do not use the `bbbc` structure any more.

3.10.4.5 Allocating the atm_uni_call_ie Structure

A convergence module can allocate as many instances of the structure as are needed. The only valid way to allocate storage for and initialize the `atm_uni_call_ie` structure from a convergence module is to use the `atm_cmm_alloc_ie` function call.

IE structures are allocated in contiguous virtual memory so that convergence and signaling modules can access them as an array of structures. Each IE structure in the array can be accessed as an index from the value returned from this call. For example, the following code fragment shows how to access the *n*th structure in the array:

```
atm_uni_call_ie_p upp, pp;
register int i;

        /* Allocate 3 IE's */
upp = atm_cmm_alloc_ie(3);
        /* Make the first IE a repeat indicator */
p = upp;
p->ie_type = ATM_IET_REPEAT;
ATM_IE_SETVAL(p->ie.bbrepeat.bbri,2);
```

```

        /* make the next two BLLI */
p = upp+1;
p->ie_type = ATM_IET_BLLI;
ATM_IE_SETVAL(p->ie.bb_low_layer,layer2proto,...);
        /* fill in structure */
p = upp+2;
p->ie_type = ATM_IET_BLLI;
ATM_IE_SETVAL(p->ie.bb_low_layer.layer2proto,...);
        /* fill in structure */

```

If you are using non-UNI signaling protocols, the convergence modules that use these protocols might not be able to use this structure directly, but must still allocate memory for passing call setup information between the signaling and convergence modules. To do this, use the `atm_cmm_alloc_ie` routine to allocate memory and assist the CMM in properly freeing the memory when the connection is destroyed. The allocating module is free to use memory returned from this call, with the following restrictions:

- Memory is allocated in multiples of the structure size so the minimum number of `atm_uni_call_ie` structures that provide enough storage must be allocated.
- The only address that convergence and signaling modules are permitted to use to reference this memory is the address returned from the `atm_cmm_alloc_ie` call.

To allocate some arbitrary amount of memory for use in call setup, use the following algorithm:

```

register char *p;
register int size;

size = (bytes_to_alloc + (sizeof(struct atm_uni_call_ie)-1)) /
        sizeof(struct atm_uni_call_ie);
p = (char *)atm_cmm_alloc_ie(size);

```

Typically, the convergence protocol module frees all UNI signaling structures associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails) the allocating module (either the convergence or signaling module) might have to free memory it has allocated. Convergence modules should not free memory for this structure if the `atm_cmm_connect` or related call succeeds because the CMM will free the memory.

If the `connect` or related call fails, the CMM does not free the memory for structures passed in as arguments since it permits the calling routine to retain the memory for future tries. In this case, the convergence module must free the memory for these structures if they will not be retained for

future call attempts. Modules can call the `atm_cmm_free_ie` function call with the value returned from `atm_cmm_alloc_ie` to free memory.

3.10.4.6 Setting Fields in the `atm__uni_call_ie` Structure

When a protocol convergence module needs to create a new connection, the module allocates and fills in an `atm_call_ie` structure array and passes this array to the CMM as part of the `atm_cmm_connect` function call. The convergence module must completely set all information in each IE that it creates; there are no default values. All IEs must be allocated using the routines described in Section 3.10.4.5 and must be referenced as an array of structures. Only the value returned from the `atm_cmm_alloc_ie` routine can be used to pass references to the IE array to the CMM. Never pass a pointer to an arbitrary element within the array to the CMM.

Since many of the IE fields are optional, each field explicitly set by a convergence module or by an incoming SETUP message must be explicitly flagged. Table 3–4 describes macros that provide the only valid means to read and write values to and from the `atm__uni_call_ie` structure.

Table 3–4: Information Element Macros

Macro	Meaning
<code>ATM_IE_SETVAL(Field,Value)</code>	Sets a value in an IE field. This sets the field to the specified value and marks the field as valid; any previous field value in the field is destroyed. The <code>Field</code> argument is a C statement that references the field to be written, and <code>Value</code> is the value to write to the field.
<code>ATM_IE_ISVALID(Field)</code>	Determines if the referenced field holds valid data (was explicitly specified by the entity that created the IE).

Table 3–4: Information Element Macros (cont.)

Macro	Meaning
<code>ATM_IE_GETVAL(Field)</code>	Reads values from an IE field. This returns 0 if the field does not contain valid data. To distinguish between a valid value of 0 and an unspecified field, use the <code>ATM_IE_ISVALID()</code> macro to determine the field's state.
<code>ATM_IE_CLRVAL(Field)</code>	Deletes valid information from an IE field. This deletes the data in the field and marks the field as invalid. This is the only way to remove (unset) a value from a field.

3.10.5 The atm_ppa Structure

The ATM subsystem uses the `atm_ppa` structure to identify a unique address/interface/signaling triple in the DIGITAL UNIX ATM subsystem. The PPA specifies the calling party address on outgoing calls and specifies the called party addresses to which the ATM subsystem will respond on incoming calls. The `atm_ppa` structure has the following characteristics:

- The CMM maintains exactly one PPA structure for each address or driver combination configured on the system.
- Only the CMM can change the contents of the PPA structure.
- Convergence modules can hold references to the PPA structures until they receive a notification that the PPA has been deleted.
- The structure is always referenced as type `atm_ppa_p`.

Table 3–5 lists those members of the `atm_ppa` structure, with their associated data types, that modules might reference.

Table 3–5: The atm_ppa Structure Members

Member Name	Data Type
<code>driver</code>	<code>atm_drv_handle_t</code>
<code>sig</code>	<code>atm_sig_handle_t</code>
<code>ppas_id</code>	<code>void *</code>
<code>ton</code>	<code>unsigned char</code>
<code>anpi</code>	<code>unsigned char</code>
<code>addrlen</code>	<code>unsigned char</code>
<code>address[ATM_PPA_MAX_ADDR]</code>	<code>unsigned char</code>

Table 3–5: The atm_ppa Structure Members (cont.)

Member Name	Data Type
uni	atm_uni_type_t
type	atm_ppa_type_t
esi_arg	void *

3.10.5.1 The driver Member

The `driver` member indicates the driver or physical interface to which this PPA belongs. Since each interface can have any number of PPAs (one for each registered address plus one PVC PPA), this member identifies the underlying device. This value is significant only to the ATM subsystem, but it can be used by convergence modules to determine if two PPAs are associated with the same physical interface.

3.10.5.2 The sig Member

The `sig` member indicates the signaling module to which the PPA belongs; the signaling module handles connection management for that PPA.

3.10.5.3 The ppas_id Member

The `ppas_id` member is a value provided by the signaling module that created the PPA for it to use in identifying or managing the PPA. The signaling module provides this value when registering the PPA with the CMM. The CMM does not use or modify this value.

3.10.5.4 The ton Member

The `ton` member is a value that specifies the endpoint's address type. These bits are identical to the type of number field in the Called Party Number IE (right justified).

3.10.5.5 The anpi Member

The `anpi` member is the address or numbering plan identification information for the address. These bits are identical to those in the corresponding field of the Called Party Number IE.

3.10.5.6 The addrlen Member

The `addrlen` member contains the length (in bytes) of the endpoint address in the address array.

3.10.5.7 The address Member

The `address` member is an array that contains one complete ATM address for the PPA. This is the fully registered address that contains either a valid E.164 address or a valid AESA (with both network and ESI parts).

3.10.5.8 The uni Member

The `uni` member identifies the type of UNI interface to which this PPA belongs. The types of UNI interface are as follows:

- `ATM_UNI_PRIVATE` — A private UNI interface
- `ATM_UNI_PUBLIC` — A public UNI interface

In DIGITAL UNIX Version 4.0, all ATM interfaces are private UNI interfaces.

3.10.5.9 The type Member

The `type` member identifies the PPA type. The PPA types are as follows:

- `ATM_PPA_PUBLIC` — Global PPAs that all convergence modules share.
- `ATM_PPA_PRIVATE` — PPAs that are private to a specific convergence module. The PPA was created using an ESI provided by that convergence module.

3.10.5.10 The esi_arg Member

The `esi_arg` member stores the convergence module private argument passed to the CMM when a private ESI is registered. Convergence modules use this information to track which PPAs are created from specific ESIs. This member is valid only for private PPAs, but contains NULL in public ESIs.

3.10.6 The atm_esi Structure

The `atm_esi` structure accommodates non-UNI ESI or equivalent address information for signaling modules that have similar requirements in forming full network addresses. Only the signaling module cares about the actual structure of the address since it is the entity that interprets this information for registration with the network. Of course, any entity that creates an ESI must make the ESI conform to the format that the signaling module uses. In addition, the `atm_esi` structure has the following additional characteristics:

- The pointer to the structure is maintained as the system-wide reference to the structure.

- Only the CMM allocates and frees the structure. Only the CMM may modify the nonprivate structure members.
- Non-UNI signaling protocols are free to use the ESI data in any way that is appropriate for the protocol, including setting full addresses from the `atm_esi` structure.

Table 3–6 lists those members of the `atm_esi` structure, with their associated data types, that modules might reference.

Table 3–6: The `atm_esi` Structure Members

Member Name	Data Type
<code>esi[ATM_MAX_ESI]</code>	unsigned char
<code>esilen</code>	unsigned int
<code>driver</code>	<code>atm_drv_handle_t</code>
<code>sigp1</code>	void *
<code>sigp2</code>	void *

3.10.6.1 The `esi` and `esilen` Members

The `esi` member is an array that holds `esilen` bytes of the ESI. The length of the ESI must be less than `ATM_MAX_ESI`. For UNI signaling, the ESI length must be 6 bytes.

3.10.6.2 The `driver` Member

The `driver` member is the driver handle for the interface on which the ESI is configured. Each ESI is associated with exactly one interface. However, different interfaces can have the same ESI values, but they must be in separate ESI structures.

3.10.6.3 The `sigp1` and `sigp2` Members

The `sigp1` and `sigp2` members are for the private use of signaling modules to keep internal information about the ESI. The CMM does not modify these members.

3.10.7 The `atm_cause_info` Structure

The `atm_cause_info` structure enables the CMM to keep track of all errors that occur on a VC or endpoint as well as record information that may be used to log activity on a VC or endpoint. The structure stores the following types of information:

- Cause information

This can be any VC and endpoint errors that are made visible to the network and to the other end of the connection through a Cause IE. Cause information also contains information of local significance only (such as text describing the error), but must contain a valid ATM cause code.

- Logging information

This can be any VC and endpoint activity information that is simply logged on the local system and not converted to Cause IE. Logging information provides information about VC or endpoint activity that a system or network administrator can view on the local system by using ATM administration tools.

The `atm_cause_info` structure has the following additional characteristics:

- All ATM modules use this structure to read cause and logging information from a VC or endpoint.
- Only the CMM allocates the structure.
- All cause and logging information retrieval functions return a pointer to this structure.
- The structure is not used for setting cause and logging information.
- The CMM writes cause and logging information when logging routines are called. The structure contents are not directly modified.

Table 3–7 lists those members of the `atm_cause_info` structure, with their associated data types, that modules might reference.

Table 3–7: The `atm_cause_info` Structure Members

Member Name	Data Type
<code>cause</code>	<code>atm_error_t</code>
<code>location</code>	<code>atm_location_t</code>
<code>module_name</code>	<code>char *</code>
<code>reason</code>	<code>char *</code>
<code>diag_length</code>	<code>unsigned char</code>
<code>diag[ATM_DIAGNOSTIC_LENGTH]</code>	<code>unsigned char</code>

3.10.7.1 The `cause` Member

The `cause` member specifies the type of ATM error that occurred.

3.10.7.2 The location Member

The `location` member specifies the ATM location code indicating where the error occurred (that is, user and network).

3.10.7.3 The module_name Member

The `module_name` member specifies the ATM module in which the error occurred.

3.10.7.4 The reason Member

The `reason` member is a string that describes the error.

3.10.7.5 The diag_length Member

The `diag_length` member specifies the length of the diagnostic information.

3.10.7.6 The diag Member

The `diag` member contains diagnostic information from the diagnostic field in the Cause IE.

Device Driver Interface

The ATM device driver interface enables ATM device drivers to communicate with the ATM subsystem. It is a registration-based interface that dynamically configures itself at system boot time. The ATM device drivers resemble other DIGITAL UNIX device drivers, with the following exceptions:

- The interface to the ATM subsystem is different than the current character, block, or networking device driver interfaces.
- ATM drivers must register with the ATM Connection Management Module (CMM).
- ATM drivers provide no `read`, `write`, `ioctl`, or `select` routine since all these functions are handled through the CMM.
- ATM drivers must provide a probe and attach routine for the operating system to call to determine if the device is present and to attach the device to the operating system.

Like networking drivers, ATM device drivers do not require an entry in the system's `cdevsw` table.

The ATM device driver interface enables device drivers to:

- Register the device driver
- Receive data packets and cells
- Report errors

This chapter describes each of these tasks, the function calls involved, and the relevant data structures that device driver writers can use. Appendix A contains a reference page for each device driver interface routine.

4.1 Registering the Device Driver

When a DIGITAL UNIX system is booted, it calls each device driver's probe routine to determine if the device that the device driver controls is present on the system and is functional. If the probe routine finds a device, it uses the return value to indicate this to the kernel. The kernel then calls the device's attach routine to attach itself to the system.

When a device driver's attach routine is called, it calls the `atm_cmm_register_dd` routine before returning from the attach. This registers the device driver with the CMM.

4.2 Receiving Data Packets and Cells

When an ATM device driver receives data packets and cells from the ATM network, it calls the `atm_cmm_receive` routine to pass the data to the CMM. When the CMM receives the data, it immediately passes the data to the convergence module that owns the virtual circuit (VC). The CMM does not queue the data. That way, the convergence modules receive the data at their input functions in the interrupt context; the driver should do all receive processing in interrupt context for efficiency.

After passing data to the CMM, the driver must not reference any mbufs in the mbuf chain again. If the driver must allocate private storage for data (rather than allocating from the system mbuf pool), the driver must provide an appropriate free routine and set the `m_ext` structure in the mbuf appropriately. The data mbuf chain is deallocated only when the protocol stack has finished referencing the data.

When an ATM device driver receives operations and maintenance (OAM) cells (nondata cells) from the ATM network, it calls the `atm_cmm_oam_receive` routine to pass the cells to the CMM for processing. Device drivers should pass all OAM cells to the CMM.

4.3 Reporting Errors

When an ATM device driver detects errors on VCs or other interface failures, it calls the `atm_cmm_error` routine to report the error to the CMM. When the CMM receives the error report, it recovers or shuts down the VC in error. The driver does not need to perform any other actions. If the VC must be destroyed, the CMM calls the driver through the driver management interface to deactivate or destroy the VC. If the error indicates an interface failure, the CMM tears down VCs on the interface.

4.4 Using ATM Device Driver Interface Structures

The ATM device driver interface uses the following structures exclusively:

- `atm_drv_params` — The device parameters and statistics structure
- `atm_queue_param` — The queue parameter structure

4.4.1 The atm_drv_params Structure

The CMM and convergence modules need to know the capabilities of each physical interface. Since this information is maintained by the driver for each interface, the system uses the `atm_drv_params` structure to communicate interface parameters and statistics between modules. The CMM typically queries this information from device drivers and then passes it to other ATM modules on request. The CMM and convergence modules can request this information at any time, and there is no limit on the number of times the information can be queried. Table 4-1 lists those members of the `atm_drv_params` structure, with their associated data types, that device drivers might reference.

Table 4-1: The atm_drv_params Structure Members

Member Name	Data Type
<code>name</code>	<code>char *</code>
<code>unit</code>	<code>unsigned int</code>
<code>type</code>	<code>atm_interface_t</code>
<code>num_vc</code>	<code>unsigned int</code>
<code>max_vcib</code>	<code>unsigned int</code>
<code>max_vpib</code>	<code>unsigned int</code>
<code>max_vci</code>	<code>unsigned int</code>
<code>max_vpi</code>	<code>unsigned int</code>
<code>sent</code>	<code>unsigned long</code>
<code>received</code>	<code>unsigned long</code>
<code>dropped</code>	<code>unsigned long</code>
<code>num_vci</code>	<code>unsigned int</code>
<code>num_vpi</code>	<code>unsigned int</code>
<code>hard_mtu</code>	<code>unsigned int</code>
<code>nqueue</code>	<code>unsigned int</code>
<code>flowcontrol</code>	<code>unsigned int</code>
<code>rates</code>	<code>atm_vc_services_t</code>
<code>capabilities</code>	<code>unsigned int</code>

Table 4–1: The atm_drv_params Structure Members (cont.)

Member Name	Data Type
numid	unsigned char
ids[6][1]	unsigned char

4.4.1.1 The name member

The `name` member specifies a character string that represents the name by which the device is known to the system. Management programs use this string when displaying information about the device. Do not include the character representation of the unit number in this name (for example, specify ATM rather than ATM0).

4.4.1.2 The unit Member

The `unit` member specifies a unit number for the device for those instances where the driver controls more than one device. The unit numbers start at 0.

4.4.1.3 The type Member

The `type` member specifies the type of interface that the device uses.

4.4.1.4 The num_vc Member

The `num_vc` member specifies the maximum number of VCs the driver can have open simultaneously.

4.4.1.5 The max_vcib and max_vpib Members

The `max_vcib` and `max_vpib` members specify the largest value the driver permits for VCI and VPI values, respectively.

4.4.1.6 The max_vci and max_vpi Members

The `max_vci` and `max_vpi` members specify the maximum number of VCIs and VPIS, respectively, that the driver can support.

4.4.1.7 The sent Member

The `sent` member is a counter of the total number of cells sent by the driver since it was last brought on line.

4.4.1.8 The received Member

The `received` member specifies the total number of cells received by the driver since it was last brought on line.

4.4.1.9 The dropped Member

The `dropped` member specifies the total number of cells dropped by the driver since it was last brought on line.

4.4.1.10 The `num_vci` and `num_vpi` Members

The `num_vci` and `num_vpi` members specify the number of VCIs and VPIs, respectively, that the interface is currently configured to process.

4.4.1.11 The `hard_mtu` Member

The `hard_mtu` member specifies the maximum packet size (in bytes) the driver can accommodate when processing cooked packets (for AAL3/4 and AAL5 only).

4.4.1.12 The `nqueue` Member

The `nqueue` member specifies the total number of queues that the interface supports. If the driver supports more than one queue, the CMM assumes the driver is capable of permanently associating VCs to queues for the lifetime of the VC.

4.4.1.13 The `flowcontrol` Member

The `flowcontrol` member specifies the types of flow control the driver supports. Valid values are `ATM_FLOW_NONE`, `ATM_FLOW_STD`, and `ATM_FLOW_VENDOR`. The CMM uses this information to assign VCs to multiple queues based on connection service parameters to achieve the best performance and fairness in cell transmission.

4.4.1.14 The `rates` Member

The `rates` member specifies an `atm_vc_services` structure that the driver has set to indicate the maximum cell rates that the interface supports for each cell rate class. These values include any amount the driver can overcommit for the interface. The CMM uses these values to determine if a connection can be granted its requested bandwidth. The other members of the `atm_vc_services` structure are unused in this context.

4.4.1.15 The capabilities Member

The `capabilities` member specifies miscellaneous driver capabilities. It is a bit mask that the driver uses to specify functions that the driver can or cannot perform.

The CMM currently uses the following capabilities:

Capability	Description
ATM_DC_DOESPTI	Indicates whether the driver keeps track of the Payload Type Indicator (PTI) bits across all cells of cooked packets (when the driver or hardware reassembles the packets from ATM cells). This capability keeps track of the congestion indication bit in the PTI field of the ATM cell header. If this bit is set, the CMM expects the PTI information provided to the <code>atm_cmm_receive</code> call to be valid. If this bit is clear, the CMM ignores the PTI information on received packets.
ATM_DC_DOESGFC	Indicates whether the device driver supports passing Generic Flow Control (GFC) bits to the CMM and to convergence layers through the CMM. The use of the GFC bits is not finalized. Therefore, the CMM ignores any GFC bits.
ATM_DC_DOES_AAL5	Indicates that the driver or adapter directly implements AAL5. The driver or adapter adds the AAL5 trailer to outgoing packets (including cyclical redundancy check (CRC) generation) and splits the packets into cells. The driver or adapter will reassemble incoming cells into an AAL5 packet and check the AAL5 trailer CRC. If this bit is set, convergence modules can send or receive packets and do not have to implement the AAL5 SAR functions. If this bit is not set, convergence modules must implement all AAL5 SAR functions and send or receive full ATM cells, if they want to send and receive AAL5 cells.

Capability	Description
ATM_DC_DOES_AAL3	Indicates that the driver or adapter directly supports AAL3/4. The driver or adapter adds the AAL3/4 headers and trailers (including CRC) and splits the packets into cells. The driver or adapter reassembles incoming cells into an AAL3/4 packet and checks the CRC. If this bit is set, convergence modules can send or receive packets and do not have to implement the AAL3/4 SAR functions. If this bit is not set, convergence modules must implement all AAL3/4 SAR functions and send or receive full ATM cells, if they want to send and receive AAL3/4 cells.
ATM_DC_DOES_RAW	Indicates that the driver or adapter is capable of sending and receiving raw ATM cells. The driver or adapter accepts raw ATM cells (full 53-byte cells) for transmission and passes incoming cells to the CMM as received. A driver can support both raw and AAL5 or AAL3/4 handling simultaneously. If a driver does not support raw cells, convergence modules cannot send arbitrary cells through the interface controlled by the driver. Since most drivers do not support AAL1, AAL2, or AAL3/4 directly, these must be handled by convergence modules as raw cells; the convergence modules must perform any needed SAR functions.

4.4.1.16 The numid Member

The `numid` member specifies the number of 48-bit Media Access Control (MAC) addresses configured on the device's address read-only memory (ROM). The driver must allocate an `atm_drv_params` structure large enough to hold all the 48-bit MAC addresses by allocating space as follows:

```
sizeof(struct atm_drv_params) + numid*6 bytes
```

4.4.1.17 The ids Member

The `ids` member specifies an array that holds the 48-bit MAC addresses. Each address appears in the array sequentially starting at `ids[6*n]`, where `n` is the address number. The CMM uses these addresses as the end system identifiers (ESI) when registering with the switch.

4.4.2 The atm_queue_param Structure

The `atm_queue_param` structure is an argument to the `ATM_DRVMGMT_RAWPARAM` ATM management command. See the description for the `atm_cmm_register_dd` routine in Appendix A for more information.

Table 4-2 lists those members of the `atm_queue_param` structure, with their associated data types, that device drivers might reference.

Table 4-2: The `atm_queue_param` Structure Members

Member Name	Data Type
<code>vc</code>	<code>atm_vc_p</code>
<code>qlength</code>	unsigned int
<code>qtime</code>	unsigned int
<code>flags</code>	unsigned int

4.4.2.1 The `vc` Member

The `vc` member specifies the VC for which the parameters are being set.

4.4.2.2 The `qlength` Member

The `qlength` member specifies the number of cells to be queued for passing in a single mbuf chain. The driver waits until this number of cells has been received before passing the cells to the CMM. Larger values reduce the per-cell overhead, but increase latency and the amount of memory consumed to buffer the cells.

4.4.2.3 The `qtime` Member

The `qtime` member specifies the maximum interval between cells (in 10-millisecond intervals) before queued cells are passed to the CMM. This value enables the driver to control latency. A value of 0 specifies an infinite amount of time between cells. The interaction between the `qlength` and `qtime` values is similar to that in the TTY subsystem. Basically, the driver accumulates cells until `qlength` cells are received or until there has been more than `qtime` ticks since the last cell was received.

4.4.2.4 The `flags` Member

The `flags` member specifies one of the following behaviors from the driver:

Behavior	Definition
ATM_QP_STAMP	<p>Instructs the driver to time-stamp all incoming cells and place the time-stamp immediately after the cell data in the mbuf. The time is a 64-bit value taken from the systems free-running clock and has a resolution of 10 nanoseconds. These values are useful only for determining the approximate cell interarrival time, by subtracting the time-stamp of cell n from the time-stamp of cell $n+1$. The time-stamp length is included in the mbuf length (time-stamped cells have an mbuf length of 61 rather than 53). See Section 3.4 for more information on data formats.</p>
ATM_QP_EFLAG	<p>Instructs the driver to insert 0-length mbufs to indicate that a cell was dropped due to an error or insufficient resources. If time-stamping is also enabled, the mbuf contains only a 64-bit time-stamp (the mbuf length is 8 bytes). Only cells lost in a manner detectable by the driver are flagged. Cells lost on the network are not tagged. Protocols using constant bit rate (CBR) can determine if a cell was lost on the network or at the sending station by a gap in the cell interarrival time. See Section 3.4 for more information on data formats.</p>

5

Signaling Module Interface

The DIGITAL UNIX ATM subsystem signaling module interface enables signaling protocol modules to communicate with the ATM subsystem for connection management only. The primary signaling module provided with the base system is the UNI 3.0/3.1-compliant signaling module. However, you can add new signaling modules if the modules do not conflict with each other (such as in the use of well-known virtual circuits (VCs)). Once configured, any convergence module can use a signaling module on a per-call basis.

Note

New signaling modules should make no assumptions about which convergence modules use them, though the convergence modules may need to know about specific signaling modules.

The signaling module interface does not provide functions that allow the signaling module to exchange data with an ATM switch. To exchange data, signaling modules must also register as convergence modules for purposes of creating signaling VCs, usually permanent virtual circuits (PVCs), and of exchanging signaling data with the network.

The ATM signaling module interface enables signaling modules to:

- Register the signaling module
- Receive, reply to, activate, release, and delete connections
- Restart a VC
- Drop endpoints
- Add and delete physical points of attachment (PPA)
- Report a connection failure, a completed restart, a completed status enquiry, and Management Information Base (MIB) access
- Request VC and endpoint status

This chapter describes each task, the function calls involved, and the relevant data structures that signaling module writers can use. Appendix A contains a reference page for each signaling module interface routine.

5.1 Registering the Signaling Module

When a signaling module is initialized, the module must use the `atm_cmm_register_sig` function call to register with the Connection Management Module (CMM). Once registered, the CMM can use the signaling module and make it available to convergence modules.

Signaling modules should register as both signaling and convergence modules. When registering as a convergence module, a signaling module must set up its signaling VCs to send and receive signaling messages. The CMM does not provide special facilities for signaling VCs, but treats them like any other VC. See Section 7.6 for a description of signaling VC initialization.

5.2 Receiving a New Call

When a signaling module receives a new call, it uses the `atm_cmm_new_call` function call to notify the CMM of the new call. The CMM determines whether the call should be accepted. This CMM function call is nonblocking and returns with an indication about the disposition of the call.

5.3 Reporting a VC Activation

When a signaling module receives notification from the switch that a new connection has been completed and a new VC activated, the signaling protocol module uses the `atm_cmm_reply` function call to notify the CMM. The CMM passes the notification to the device driver and protocol convergence module.

This call notifies the CMM of both the activation of point-to-point connections and of the addition of endpoints in point-to-multipoint connections.

5.4 Activating a Connection

When a signaling module receives the necessary indication from the network that the circuit is connected, the signaling module uses the `atm_cmm_activate_con` function call to inform the CMM that the connection is ready to carry data. The CMM enables the VC locally.

Note

Do not make this call before circuit connection.

5.5 Reporting a Connection Failure

When the CMM makes a connection request to a signaling module, the call to the remote system might fail for some reason. Since call creation is an asynchronous operation, the signaling module might encounter errors during call processing after the `xxx_setup` call has returned. In these cases, the signaling module uses the `atm_cmm_con_failed` function call to notify the CMM of the call failure and the reason for the failure.

5.6 Releasing a Connection

When a signaling module receives a request to tear down a connection from the network or endpoint, the module uses the `atm_cmm_con_release` function call to notify the CMM that the connection will be released. The CMM makes the connection unavailable for transmission, initiates the teardown of the referenced connection and all associated endpoints, and awaits notification that the VC has been torn down.

5.7 Dropping an Endpoint

When a signaling module receives a request to drop an endpoint from a connection, the module uses the `atm_cmm_ep_dropped` function call to notify the CMM that the endpoint must be dropped. The CMM then notifies the convergence module that owns the endpoint's VC that the endpoint has been dropped and deletes the endpoint. If the last endpoint associated with a VC is dropped, the CMM initiates the release of the VC.

5.8 Deleting a Connection

When a signaling module receives confirmation of a connection's release from the switch, the module uses the `atm_cmm_con_deleted` function call to notify the CMM that the connection no longer exists. The CMM holds the resources associated with the connection for a brief period of time, then releases them, giving all incoming queues time to clear. The CMM also notifies convergence modules that the connection has been released.

5.9 Restarting a Virtual Circuit

When a signaling module receives a RESTART request, the module uses the `atm_cmm_restart` function call to pass all the appropriate information to the CMM. The CMM then initiates a restart of the indicated VC(s) before returning to the signaling module.

5.10 Reporting a Completed Restart

After the CMM requests a restart, the signaling module uses the `atm_cmm_restart_ack` function call to notify the CMM when the restart has completed. Once notified, the CMM returns the indicated VCs to the NULL state and frees their resources.

5.11 Reporting a Completed Status Enquiry

After the CMM requests a status enquiry of a VC, the signaling module uses the `atm_cmm_status_done` function call to notify the CMM when the enquiry has completed. Once notified, the CMM either examines the enquiry data itself or passes it to the convergence module that requested the enquiry.

5.12 Requesting Endpoint Information

Since signaling modules can receive messages for existing VCs at any time, the modules use the `atm_cmm_findaddr` function call to request endpoint and VC information from the CMM. The CMM maintains all state information about each VC and calls (endpoints) associated with the VC.

Signaling modules can call this routine at any time to resolve a reference to a connection endpoint. Once the reference is resolved, the signaling module can access and modify structures as necessary as long as locking conventions are followed.

5.13 Adding a PPA

Signaling protocols usually perform some protocol registration-specific registration with the switch, including registering local addresses. When a signaling module creates a new address (creating a new PPA), the module uses the `atm_cmm_new_ppa` function call to inform the CMM that this new PPA exists. A convergence module can use the PPA to make and receive calls.

For example, when a UNI 3.0/3.1 signaling module is informed of a new address prefix, through the Interim Local Management Interface (ILMI), that has been created on the switch, the module combines the new prefix with existing end system identifiers (ESIs) to form a new set of addresses for the new prefix. Then, the signaling module tells the CMM about each of these new addresses.

5.14 Deleting a PPA

When a signaling protocol, in cooperation with a switch, deletes an address from the list of recognized addresses on an interface, it uses the `atm_cmm_del_ppa` function call to inform the CMM that the deleted PPA associated with the address is no longer valid. The CMM then informs convergence modules bound to the PPA that the address is no longer valid and initiates a teardown of all VCs associated with the address.

All PPAs, except for the PVC PPAs, are owned by a signaling module. That is, a signaling module is always responsible for the creation and deletion of a PPA. This is required since the registration of addresses with a switch is handled entirely by signaling protocols. Also, PPAs can be deleted because of actions on the network that are completely unrelated to the local system. Because of this, the CMM does not automatically delete PPAs when an interface is taken down or loses its connection to the switch. The CMM responds to an interface shutdown by deleting the PVC PPA. Signaling modules will delete PPAs when it is appropriate for their protocols to do so (such as when they lose communications with the switch).

See Section 3.6.1 for more information on PVC PPAs.

5.15 Reporting Completed MIB Access

Sometimes a signaling module's Management Information Base (MIB) access function call can block because of the need to contact some other network management entity. When the signaling module returns from the MIB access call, it uses the `atm_cmm_mib_response` function call to notify the CMM of the completion of a MIB access and report any retrieved values.

5.16 Requesting VC Status

When a signaling module needs to access status information about a VC that is currently in service, based on the virtual path identifier (VPI) and virtual channel identifier (VCI), it calls the `atm_cmm_vc_get` function call. The call returns a reference to a VC.

5.17 Using the `atm_oid` Structure

The ATM signaling module interface uses the `atm_oid` structure exclusively to communicate MIB objects and object instances between signaling modules and the CMM. Table 5-1 lists those members of the `atm_oid` structure, with their associated data types, that signaling modules might reference.

Table 5–1: The atm_oid Structure Members

Member Name	Data Type
len	int
id[1]	int

The `len` member specifies the length of the `id` member. The `id` member specifies the Object ID (OID).

6

Convergence Module Interface

The ATM convergence module interface enables all kernel-level protocol convergence modules (convergence modules, for short) to interface kernel networking protocol stacks to the DIGITAL UNIX ATM subsystem. The ATM convergence module interface is a registration-based interface that provides a set of ATM primitives that convergence modules can use to manage virtual circuits (VCs) and transfer data.

The ATM convergence module interface enables convergence modules to:

- Register and unregister the convergence module
- Receive connection, data, and exception notification from the CMM
- Connect to the ATM Module Management Interface (MMI)
- Request interface parameters, endpoint connection state information, and VC statistics
- Add an endpoint to and drop an endpoint from a connection
- Transmit data on an established VC
- Modify VC parameters
- Request a connection to a remote system and a connection teardown
- Bind to and unbind from a PPA
- Accept and reject an incoming call
- Add and delete ATM addresses

This chapter describes each task, the function calls involved, and the relevant data structures that convergence module writers can use. Appendix A contains a reference page for each convergence module interface routine.

6.1 Registering a Convergence Module

Before a protocol convergence module can interact with the ATM subsystem, it must use the `atm_cmm_register_cvg` function call to register itself with the Connection Management Module (CMM). The CMM then passes interface configuration information (for example, currently configured PPAs) to the convergence module, using the module's `xxx_except` routine, before the registration call returns.

Once a convergence module is registered, the CMM knows where to deliver connection notifications and various exception notifications (for example, changes in system configuration). Convergence modules can register at any time.

6.2 Receiving Data

When the CMM receives data on a VC that the convergence module owns, the CMM calls the `xxx_receive` function to pass incoming data to the convergence module. This function is declared within the convergence module.

When designing receive functions, you should remember that all data is delivered to the convergence module in an interrupt context with the processor running at the `splimp` level. The convergence module must implement a queuing policy appropriate for the convergence module's protocol. In general, convergence modules for protocols that can tolerate unspecified latency should queue the incoming data and return immediately to the CMM. Convergence modules for protocols that require bounded latencies (such as video or voice protocols) might want to perform some processing before queuing the data and returning to the CMM. In either case, the receive function must not block.

6.2.1 Receiving Exception Notifications

When an exception condition occurs on a VC, the CMM notifies the convergence module through its exception interface, `xxx_except`. This function is declared within the convergence module and is used to report exceptions, errors, and system configuration changes to the convergence modules.

Exception notifications have the following characteristics:

- Convergence modules can expect them at any time.
- They are delivered in an interrupt context.
- Exception processing must not block. If a convergence module needs to defer processing an exception notification, it must arrange for a kernel thread to be run at a later time and return immediately to the CMM.

6.2.2 Connecting to the ATM Module Management Interface

When the CMM connects a convergence module to the ATM Module Management Interface (MMI), the CMM uses the `xxx_mmi` interface. This function is declared within the convergence module. System management programs use the CMM's management interface to pass management and

configuration requests directly to convergence modules, without requiring a new `/dev` entry or other kernel modifications normally associated with creating new management and configuration interfaces. If a module does not require any external management or configuration capabilities, it does not have to register a management function with the CMM.

The MMI follows the standard DIGITAL UNIX `ioctl` call with the following exceptions:

- The device major and minor number argument is replaced by the *handle* argument.
- No user credential information is passed. Only users with root access can use the management interface.

6.3 Unregistering a Convergence Module

If a convergence module is finished receiving or making connections, it can use the `atm_cmm_unregister_cvg` function call to unregister itself from the CMM. This might be necessary in environments where protocol stacks are dynamically configured and deconfigured from the system.

Before unregistering, a convergence module must close all existing connections and unbind from all PPAs. If all connections are not closed or if PPA bindings still exist, the CMM unregisters only the convergence module's `xxx_connect` routine. The CMM will continue to call the convergence module with incoming data and exception notifications. Once the convergence module is unregistered, the CMM will not call any of its routines.

6.4 Requesting Interface Parameters

When a convergence module must access current information about an interface's physical capabilities (such as its bit rates) and the amount of specific resources left available on an interface (the remaining nonreserved sustainable bit rate), it uses the `atm_cmm_ppa_info` and `atm_cmm_bind_info` function calls to query the CMM. The CMM and device drivers keep track of this information.

Since convergence modules normally deal with either PPAs or AESA bindings, the CMM allows convergence modules to query the information from the underlying interface by specifying either the PPA or bind directly. Thus, the convergence module requires no specific knowledge of any device driver or device driver-related structures.

6.5 Reserving Resources for CBR and Pacing Circuits

If a convergence module needs to reserve resources (such as bandwidth) either prior to making an outgoing connection for a constant bit rate (CBR) or pacing circuit or prior to receiving a CBR connection, it uses the `atm_cmm_reserve_resources` function call to notify the CMM. The convergence module must apply these reserved resources within the system-specified amount of time either to an outgoing CBR or pacing connection by using the `atm_cmm_connect` function call or to an incoming CBR call by using the `atm_cmm_accept` function call.

Note

Reserved resources can be applied to point-to-point connections only.

If the resources are not applied in time, the CMM revokes the reservation and sends an `ATM_CME_RESV_EXPIRE` exception notification to the convergence module. The convergence module should remove knowledge of the reserved resources `atm_vc_services` structure that is being revoked, but should not free the `atm_vc_services` structure; the CMM frees the structure and releases the underlying resources upon return from the exception notification. In addition, the convergence module cannot apply the resources to a connection as part of handling the exception notification.

Once a reserved resource is successfully applied to a connection, the CMM transparently frees the underlying resources when the connection is torn down. No special action by the convergence module is required.

6.6 Releasing Reserved Resources

When a convergence module no longer needs to apply reserved resources to a VC and wants to release resources that it previously reserved, it uses the `atm_cmm_free_services` function call with a pointer to the `atm_vc_services` structure. This frees the `atm_vc_services` memory and releases its underlying resources.

6.7 Requesting a Connection to a Remote System

When a convergence module needs a connection to a remote system, it uses the `atm_cmm_connect` function call to make a connection request to the CMM. This initiates the exchanges between the network and the remote host to create a new connection. The convergence module must specify all

connection parameters to the CMM; these parameters are required to create the new connection.

By requesting a connection to a remote host, the convergence module owns the VC; the VC's ownership cannot be changed. The convergence module is notified of all incoming data and exceptions on the VC. Also, the convergence module is the only module that can transmit data on the VC.

Note

The DIGITAL UNIX ATM subsystem does not support the creation of outgoing point-to-multipoint calls and cannot accept more than one endpoint on incoming point-to-multipoint calls. However, the subsystem can accept one endpoint on an incoming point-to-multipoint call that originates from another vendor's UNIX system.

6.8 Adding an Endpoint to a Connection

When a convergence module must create a point-to-multipoint connection, it first uses the `atm_cmm_connect` function call to establish a call to the first party of the multipoint connection. This creates a VC and establishes the connection to the first endpoint. Then you can use the `atm_cmm_add` function call at any time to add new endpoints to the connection. You can add endpoints only to connections that were created with the `ATM_CT_PTM` flag set.

See Chapter 7 for more information on connections.

6.9 Requesting a Connection Be Torn Down

When a convergence module no longer requires a connection to a remote host, it uses the `atm_cmm_release` function call to request that the CMM tear down the connection. This initiates the release negotiation with the remote host, and will eventually lead to the connection being torn down and all resources released.

A convergence module can either request that a specific endpoint be disconnected (referenced by an `atm_addr` structure) or that a VC be disconnected. These operations are equivalent on point-to-point connections; disconnecting only the endpoint results in the VC being torn down. If a convergence module requests that a point-to-multipoint VC be disconnected, the CMM first disconnects all endpoints associated with the VC. This function simplifies the task of disconnecting multipoint VCs and permanent virtual circuits (PVCs).

6.10 Dropping an Endpoint from a Connection

When a convergence module must drop a connection to an endpoint, the module calls the `atm_cmm_drop` function. The endpoint can be associated with either a point-to-point connection or a point-to-multipoint connection. When the last endpoint associated with a VC is dropped, the VC is torn down. You use this function to manage multipoint VCs, but you can also use it to initiate the teardown of point-to-point VCs.

6.11 Transmitting Data on an Established VC

When a convergence module must transmit data on an established VC, the module calls the `atm_cmm_send` function call. The CMM then passes the data to the appropriate device driver. Although the CMM does not queue outgoing data, the device driver might. Therefore, the successful return from this function call does not imply that the data was actually transmitted. In fact, the data could be discarded in the driver or on the network. The convergence module is not notified if the data is dropped locally.

6.12 Modifying VC Parameters

After VCs are created, convergence modules can use the `atm_cmm_vc_control` function call to modify some VC parameters to make the transfer of data more efficient or to control VC aging. You cannot modify the quality of service (QoS) and circuit bandwidth parameters.

6.13 Requesting Endpoint Connection State Information

When a convergence module needs the connection state information for an endpoint updated, the module calls the `atm_cmm_enquery` function call. This initiates a connection enquiry to the endpoint that results in the ATM address structure for the endpoint being updated with the latest status information. This function may be called any time a connection is active.

6.14 Binding to a PPA

When a convergence module must make outgoing calls or receive incoming calls being made to one of the configured local ATM addresses, the module calls the `atm_cmm_ppa_bind` function. This call informs the CMM that it will function as a network services user for a specific address and selector value.

Convergence modules are notified when a PPA is configured on the system (when an address is registered with a switch and is made known to the

ATM network). Convergence modules can then bind to the PPA to create a network service user endpoint, which uniquely identifies the convergence module on the network. Once bound, there is a unique ATM End System Address (AESAs) associated with the bound convergence module. The AESA and the bind point identify the service on the network and local system. Convergence modules will be informed only of incoming calls on PPAs to which they have bound and can place calls only through a bind point. The CMM handles calls only to AESAs. If the AESA specified by the called party address of an incoming call does not exist, the CMM rejects the call. See Section 3.6 for more information on PPAs and AESAs.

By binding to a PPA, a convergence module is creating an AESA and uniquely identifying its service on the network. Once this is done, incoming calls can be routed to the convergence module, and the convergence module can make outgoing calls. All call activity is directed at a bind point and thus to a specific convergence module.

When binding to a PPA, a convergence module can either specify a selector value to use in creating the AESA address, or it may allow the CMM to assign it a selector value. In both cases, the selector value must specify a unique endpoint on the PPA.

Note that a convergence module can bind up to 256 times to a PPA (as long as all its selector values are unique), and may bind to as many PPAs as is necessary to provide its service.

6.15 Receiving a Connection Notification

When a connection request arrives on the system, a signaling protocol module notifies the CMM. The CMM uses the `xxx_connect` interface to call the convergence module that is bound to the selector specified in the called party address. Only one convergence module can accept a connection. The called convergence module then examines all the connection data (the information in the `atm_addr` structure as well as any information elements (IEs) or other signaling information passed in) to determine if it is willing to accept the call.

If the convergence module accepts the call, the CMM and signaling module proceed with call setup. The VC is not active at this point. The connection and VC is owned by the accepting convergence module until the connection is destroyed; you cannot change the VC's ownership. The convergence module VC is notified of all incoming data and exceptions on the VC. Also, the convergence module is the only module that can transmit data on the VC.

Connections are not shared between convergence modules at the bind point. If convergence modules must share connections, one module must

own the connection and coordinate access to the connection with another module. Alternatively, you can layer modules on top of a multiplexor that assumes ownership of the connection.

If the convergence module rejects the call, the CMM and signaling module release the VC and notify the calling party that the call was rejected. The CMM never accepts a call without the explicit consent of a convergence module.

6.16 Unbinding from a PPA

If a convergence module must not accept an incoming connection (while awaiting a call from a specific caller) or make outgoing calls, it uses the `atm_cmm_ppa_unbind` function call to unbind its service from the PPA. Typically, convergence modules should unbind a PPA if their service is no longer available, and must unbind before unregistering with the CMM.

6.17 Accepting an Incoming Call

When a convergence module defers action on an incoming call (by returning `ATM_CAUSE_DEFER` from its `xxx_connect` routine), the module notifies the CMM as soon as it determines the disposition of the call. Typically, this is done within a set period of time, as defined by the signaling protocol, before the calling party times out on the call request.

If the convergence module determines that the call should be accepted, the module uses the `atm_cmm_accept` function call to inform the CMM that the call should be accepted and set the circuit resource parameters.

6.18 Rejecting an Incoming Call

When a convergence module defers action on an incoming call (by returning `ATM_CAUSE_DEFER` from its `xxx_connect` routine), the module notifies the CMM as soon as it determines the disposition of the call. Typically, this is done within a set period of time, as defined by the signaling protocol, before the calling party times out on the call request.

If the convergence module determines that the call should be rejected, the module uses the `atm_cmm_reject` function call to inform the CMM.

6.19 Adding a New ATM Address

When a convergence module must register a new address with the network, the module uses the `atm_cmm_new_esi` function call to supply the ESI (or other signaling protocol equivalent) portion of the address and a driver

reference to the CMM. The CMM then performs all the necessary functions needed to register the address with the network and to activate it.

When a convergence module supplies a new ESI, the signaling modules apply the ESI in a signaling protocol-specific way. Depending on the signaling protocol, the addition of a single ESI to the system could create many PPAs. All convergence modules are notified of every new PPA that is created as the result of configuring a new ESI.

Once a convergence module creates an address, only the creating convergence module, the CMM (through the system `atmconfig` program), or the network can delete the address. Other convergence modules cannot delete the address. The creating convergence module must issue the `atm_cmm_ppa_bind` call once it receives notification that the PPA(s) have been configured. If the convergence module unregisters with the CMM, all its created addresses are destroyed.

6.20 Deleting an ATM Address

When a convergence module is finished using a new address it has created, it can use the `atm_cmm_del_esi` function call to delete it. Deleting a PPA causes all VCs associated with that PPA to be closed by the CMM (both ends of the connection receive proper notification). Thus, convergence modules should delete addresses only when there are no active VCs associated with the address' PPAs.

6.21 Requesting VC Statistics

When a convergence module must retrieve current VC information from the CMM, it calls the `atm_cmm_vc_stats` function. The CMM returns the information in the `atm_vc_stats` structure.

The CMM keeps track of VC usage on a per-VC basis. This is done as part of the CMM's VC management and cannot be disabled.

6.22 Using ATM Convergence Module Interface Structures

The ATM device driver interface uses the following structures exclusively:

- `atm_vc_stats` — The virtual circuit statistics structure
- `atm_cmi_addr` — The called party address union

6.22.1 The `atm_vc_stats` Structure

The `atm_vc_stats` structure contains information on successful return from the `atm_cmm_vc_stats` function. Table 6-1 lists those members of

the `atm_vc_stats` structure, with their associated data types, that convergence modules might reference.

Table 6–1: The `atm_vc_stats` Structure Members

Member Name	Data Type
<code>bytes_in</code>	unsigned long
<code>bytes_out</code>	unsigned long
<code>packets_in</code>	unsigned int
<code>packets_out</code>	unsigned int
<code>opened</code>	struct timeval
<code>last_out</code>	struct timeval
<code>last_in</code>	struct timeval

The `bytes_in` member contains a count of the number of bytes received on the VC. This includes any ATM Adaptation Layer (AAL) protocol headers/trailers passed back by the driver.

The `bytes_out` member contains a count of the number of bytes transmitted on the VC. This does not include the AAL header/trailer bytes unless these are supplied by the convergence module.

The `packets_in` and `packets_out` members contain counts of the number of packets (or cells for raw cell VCs) received and sent, respectively.

The `opened` member contains the time-stamp of the time the VC was established. If you subtract this from the current time, the result is the time the VC has been in service.

The `last_out` and `last_in` members contain the time-stamps of the time the last data was received and sent, respectively. If you subtract either of these times from the current time, the result is the amount of time the VC has been idle in either direction.

6.22.2 The `atm_cmi_addr` Union

Convergence modules use the `atm_cmi_addr` union to pass a called party (destination) address to the CMM as part of the `atm_cmm_connect` function call. The format of this union is as follows:

```
union atm_cmi_addr {
    atm_addr_p    addr;
    struct {
        unsigned int    vci;
        unsigned int    vpi;
    };
};
```

```
    } vcn  
};
```

6.22.2.1 The `addr` Member

The `addr` member contains a pointer to an `atm_addr` structure, which is allocated by the convergence module (`atm_cmm_alloc_addr`), for creating a new switched virtual circuit (SVC).

6.22.2.2 The `vcn` Member

The `vcn` member is a structure that specifies the virtual channel identifier (VCI) and virtual path identifier (VPI) for creating a new PVC. See Appendix A for a description of the `atm_cmm_connect` function call.

7

Connections

This chapter describes the following connection tasks and the procedures for accomplishing these tasks:

- Making outgoing connections
- Accepting connections
- Controlling the aging of connections
- Releasing connections
- Creating permanent virtual circuits
- Creating signaling virtual circuits

These procedures illustrate the use of the ATM interfaces described in previous chapters.

Note

The DIGITAL UNIX ATM subsystem does not support the creation of outgoing point-to-multipoint calls and cannot accept more than one endpoint on incoming point-to-multipoint calls. However, the subsystem can accept one endpoint on an incoming point-to-multipoint call that originates from another vendor's UNIX system.

7.1 Making Outgoing Connections

Making outgoing connections consists of the following tasks:

- Making the call (for both point-to-point and point-to-multipoint connections)
- Adding parties to an existing connection (point-to-multipoint connections only)

7.1.1 Making the Call

Making the call for point-to-point and point-to-multipoint connections is identical except for one argument. When writing convergence modules, make sure the module performs the following tasks when making a call:

1. Chooses a bind point for the outgoing call.
2. Calls the `atm_cmm_alloc_addr` function to allocate an `atm_addr` structure that identifies the called party.
3. Sets all the address information associated with the called endpoint in the `atm_addr` structure.
4. Calls `atm_cmm_alloc_ie` to allocate the `atm_uni_call_ie` structure that the module uses to make the call. If you use another signaling protocol, allocate the structure needed by that protocol instead.
5. Sets all the needed information in the `atm_uni_call_ie` structure or other signaling protocol call parameters structure. If the module is to pass MTU values to the signaling module or to the other end of the connection, the module sets the values in this structure.
6. Allocates an `atm_vc_services_t` structure by calling `atm_cmm_alloc_services`.
7. Sets all the structure members in the `atm_vc_services_t` structure. For constant bit rate (CBR) circuits, set the `ATM_SERVICES_CBR` flag. For pacing circuits, set the `ATM_SERVICES_PACING` flag. For both CBR and pacing circuits, you must set the peak bit rates.

Also, set the `fmtu` and `bmtu` structure members with the largest MTU value that the module will use. The device driver uses this information to allocate resources for the VC.

8. If you want to reserve resources for CBR or pacing circuits, call the `atm_cmm_reserve_resources` routine, passing it a pointer to the `atm_vc_services_t` structure.

Note

You can apply reserved resources to point-to-point connections only.

9. If the reserve resources call returns `ATM_CAUSE_DEFER`, the `atm_vc_services` structure is considered a reserved resource reservation. The convergence module must wait for an `ATM_CME_RESV_AVAIL` exception notification before proceeding; this exception indicates that the `atm_vc_services` structure is backed by

resources. If the convergence module does not wait for the reservation to become available, it terminates the resource reservation request (and frees the `atm_vc_services_t` memory) by calling `atm_cmm_free_services`.

If the reserve resources call returns `ATM_CAUSE_GOOD`, the `atm_vc_services_t` structure is considered to be backed by resources, which can be applied to a call.

If the reserve resources call returns any other error code, the convergence module should examine the `lerrstat` structure member of the `atm_vc_services_t` structure for additional error information. If the convergence module does not continue, it must free the memory allocated in the previous steps.

10. Calls `atm_cmm_connect` with the appropriate arguments. For point-to-point connections, uses the `ATM_CT_PTP` argument. For point-to-multipoint connections, uses the `ATM_CT_PTM` argument.

Once the connect call is made, all subsequent activity on the VC is reported through the exception interface (`xxx_except`). Receive data can also begin to arrive.

Note

If the module has reserved resources and they are not applied in time, they are revoked by an `ATM_CME_RESV_EXPIRE` exception notification.

11. If the connect call returns an error code, the convergence module frees the memory allocated in the previous steps. An error might be caused by the Connection Management Module (CMM) or other modules being unable to initiate a call request.

For CBR and pacing connections that do not use resource reservation, there might be a failure to allocate the necessary resources. If the failure is due to an incorrectly set `atm_vc_services_t` structure, the `lerrstat` structure member contains local error information.

12. If the connect call returns `ATM_CAUSE_GOOD`, the call is proceeding and the convergence module can make a reference to the new `atm_vc` structure for the connection.

Section B.1 contains a sample routine that shows how to make a call for point-to-point connections and how to make the first call for point-to-multipoint connections.

After the call is initiated, the convergence module can receive the following notifications:

ATM_CME_CALL_FAILED

An error occurred in processing the call or the called party rejected the call.

The convergence module should extract the cause information from the endpoint's `atm_addr` structure and decide how to handle the error based on this information. The module should then delete all references to the `atm_addr` structure and the VC, but should not deallocate the structure memory. The CMM does this after removing all references to the call. Once the convergence module returns from the exception notification routine, it can no longer access any nonprivate structures associated with the call.

ATM_CME_EP_ACTIVE

The call is successful; the circuit to the endpoint is complete and it could receive incoming data at any time (incoming data could arrive ahead of the connection activation notification because of race conditions). The convergence module should not send any data until it receives the connection activation notification.

The call recipient could signal new MTU values. Therefore, the convergence module should examine the `atm_uni_call_ie` structure for the MTU sizes. The convergence module, however, does not have to change the `fmtu` or `bmtu` values in the `atm_vc_services` structure.

When a connection is activated, the convergence module may check the `atm_vc_services` structure referenced in the `atm_vc` structure to determine if any of its connection service parameters were changed by the called party through negotiation.

7.1.2 Adding Parties to an Existing Connection

For point-to-multipoint connections, after the call to the first party is successful, you can add parties to the VC associated with the first call and drop parties at any time. Note that since call placement is multithreaded, you can initiate calls to multiple endpoints sequentially. Convergence modules receive individual notifications for each endpoint that is connected. Thus, multiple connections can be in progress at any given time without the convergence module needing to handle synchronization issues.

The convergence module can receive exception notifications for exceptions affecting all parties on the VC or for single endpoints (an exception with a single party in the multipoint call). When writing a convergence module, you must ensure that the module can handle these exceptions properly so that it does not associate an endpoint exception affecting a specific party with all parties on a VC.

Section B.2 contains a code fragment that shows one method of adding parties to a point-to-multipoint connection.

7.2 Accepting Connections

When a remote host places a call to the local host, the following events occur on the local host:

1. The signaling protocol module on the local host receives a call request (a UNI 3.0/3.1 SETUP message or its equivalent).
2. The signaling protocol module uses the information in the signaling protocol message to construct an `atm_addr` structure and an `atm_vc_services` structure. These structures contain information about the calling endpoint and the services requested by the endpoint, respectively.
3. The signaling module uses the `atm_cmm_new_call` function call to pass the structures, the handle of the device driver on which the new VC will be attached, and a unique call reference value to the CMM.
4. The CMM examines the `atm_vc_services` structure parameters to determine if the available system resources can meet the level of service requested by the calling party. If so, the resources are reserved. If not, only available resources are reserved. The CMM calls the convergence module (using the `xxx_connect` routine) that has bound to the called party's ATM End System Address (AESAs).
5. When called, the convergence module should inspect the ATM address information and the signaling protocol information (referenced by the `setup` member of the `atm_addr` structure) to determine if the call is for the convergence module's protocol. If the call does not belong to the convergence module, it should return an error indication to the CMM. If the call belongs to the convergence module, it continues with step 6.
6. The convergence module must set the `fmtu` and `bmtu` fields in the `atm_vc_services` structure. This is how the convergence module notifies the device driver of the MTU to use for the call.
7. The convergence module should check the service parameters arguments to determine if the service parameters are acceptable. If the service parameters are insufficient for the type of service the module is providing, it can reject the call with the appropriate cause. The convergence module cannot modify the service parameters.

If the CMM has allocated the resources needed to meet the requested level of service, the convergence module should return `ATM_CAUSE_GOOD` to the CMM to accept the call. If the CMM indicates that it cannot allocate the bandwidth and the convergence module does

not apply a reserved resource to the incoming call, the convergence module should return an error indication to the CMM to reject the call.

If the convergence module supplies a reserved resource that cannot be applied to the call, the new VC is torn down, the supplied `atm_vc_services` structure and its underlying resources freed, and the convergence module sent an `ATM_CME_EP_DEAD` exception notification. The VC's cause log contains the reason why the connection was torn down.

Section B.3 contains a sample code fragment that shows how a convergence module processes an incoming call.

8. If a convergence module accepts a call, the following should occur:
 - The convergence module should use the convergence module private fields in the `atm_addr` and `atm_vc` structures to create its internal references to the new VC and endpoint.
 - Once the connection notification call (`xxx_connect`) returns with `ATM_CAUSE_GOOD`, the convergence module owns the call and owns and controls the VC. From this point, all notifications associated with the VC go to the accepting convergence module.
 - The convergence module should consider the VC valid but not active. However, data can start arriving at any time since the path the activation notification takes could be longer than the data path.
 - The convergence module should not attempt to transmit data until it receives a notification that the VC is active.
 - The CMM returns `ATM_CAUSE_GOOD` to the signaling module, indicating that the signaling module should reply to the calling party in whatever way is appropriate to accept the call.
 - The signaling module must not deallocate any memory used for structures passed to the CMM.
9. If the call is not accepted, the signaling module replies to the caller in an appropriate way to reject the call. In this case, the signaling module must deallocate all memory use for structures passed to the CMM.

7.3 Controlling the Aging of Connections

This section applies to switched virtual circuits (SVCs) only; permanent virtual circuits (PVCs) do not age. The DIGITAL UNIX ATM subsystem provides for automatic detection and elimination of unused VCs when available resources are low. This prevents a hung protocol from continuing to hold resources and ensures that system resources are allocated fairly.

When a circuit is created, convergence modules specify an initial aging algorithm in an argument to the `atm_cmm_connect` routine. The CMM then performs all connection aging based on usage statistics it collects during the lifetime of a connection. If necessary, convergence modules can use the `atm_cmm_vc_control` routine to change the aging algorithm applied by the CMM at any time. When a connection has aged, the CMM uses the convergence module's `xxx_except` routine to notify the convergence module that a connection has aged and is about to be removed from the system. This gives the convergence module a chance to prevent the connection's removal.

If the convergence module does not prevent the connection's removal, the CMM uses the normal connection release mechanism to remove the connection; the signaling module sees the release request as coming from the convergence module and the convergence module sees the request as coming from the remote endpoint. See Section 7.4 for information on releasing a connection.

Table 7-1 contains values you can use for the aging parameters in the `atm_cmm_connect` and `atm_cmm_vc_control` routines.

Table 7-1: Aging Parameter Values

Value	Meaning
<code>ATM_AGE_DEFAULT</code>	Uses the system's default aging algorithm. The default is to remove all VCs that have been inactive for the longest period of time, that have the lowest utilization (compared to the expected utilization), and whose reconnection time is expected to be less than 5 percent of the idle time.
<code>ATM_AGE_LOW</code>	Removes this VC when the system is low on free circuit identifiers and ignores its idle time and utilization.
<code>ATM_AGE_FOREVER</code>	Specifies that the circuit is not eligible for aging.
28-bit unsigned value	Specifies how long (in minutes) the circuit should be allowed to live. After the circuit is active for the specified number of minutes, it is aged from the system.
28-bit unsigned value bitwise ORed with <code>ATM_AGE_IDLE</code>	Ages the circuit when it has been inactive the specified number of minutes.

These circuit-aging algorithms provide sufficient flexibility so that individual convergence modules should not have to perform their own circuit aging. Also, centralizing circuit aging in the CMM allows aging

algorithms to take affect only when the CMM detects that an interface is low on free VCs.

7.4 Releasing a Connection

Either convergence modules or the network or an endpoint can release connections. The following sections describe what happens in either case.

7.4.1 Release by a Convergence Module

When a convergence module no longer requires a connection to an endpoint, it should release the connection to free system resources. To release a connection, a convergence module calls either the `atm_cmm_release` or `atm_cmm_drop` function. The former releases a VC and all associated endpoints; the latter disconnects only a single endpoint, releasing the VC only if all endpoints have been dropped.

The convergence module can provide a cause value and diagnostic information for the release by calling `atm_cmm_set_cause` before calling `atm_cmm_release` or `atm_cmm_drop`. For UNI-based signaling, this information is supplied to the network in a Cause IE.

When a convergence module calls `atm_cmm_release`, the CMM initiates a disconnect of all endpoints associated with the VC.

For point-to-point VCs, this initiates a release operation on the VC. The convergence module receives an `ATM_CME_EP_DEAD` notification. This informs the convergence module that the release operation is in progress on the network and that no more data will be sent on the VC.

For multipoint VCs, this initiates the dropping of all endpoints and the release of the VC. As each endpoint is dropped, the convergence module receives an `ATM_CME_EP_DEAD` notification. The convergence module must delete all references to the endpoint. Once all endpoints associated with a VC have been dropped, the VC is released. The CMM handles conditions where endpoints are in various stages of set or release when it receives this request.

When the `ATM_CME_EP_DEAD` notification is received for the last endpoint on the VC, convergence modules should delete all references to the VC. To accomplish this, DIGITAL recommends that the module maintain a use count for any VC references it maintains.

When a VC is released, the CMM notifies the device driver to which the VC is attached so that the device driver can free any resources it has allocated to the VC.

7.4.2 Release by Network or Endpoint

When the network or an endpoint initiates a connection release, the signaling protocol module receives either a release or a drop message. At that point, the signaling module calls either the `atm_cmm_con_release` or `atm_cmm_ep_dropped` function.

When a signaling module calls the `atm_cmm_con_release` function, the CMM does the following:

1. Initiates connection releases to all endpoints associated with the VC, if the VC being released is a multipoint VC.
2. Calls the device driver to which the VC is attached and passes an `ATM_DRVMGMT_DELVC` notification. The driver should delete all references to the VC and release all resources associated with the VC.
3. Calls the convergence module that owns the VC and passes an `ATM_CME_EP_DEAD` notification for each endpoint associated with the VC.
4. Frees all structures associated with the VC and returns to the signaling module.

When the signaling module calls the `atm_cmm_ep_dropped` function, the CMM calls the convergence module that owns the endpoint's VC and passes an `ATM_CME_EP_DEAD` notification. If this is the last endpoint associated with the VC, it initiates a release of the VC.

7.5 Creating Permanent Virtual Circuits

Although the ATM subsystem is designed primarily to use SVCs, you can also use PVCs. Before you create PVCs locally, the PVCs must already be set up on the network. You can use either of the following methods for setting up PVCs:

- The `atmconfig` program (see `atmconfig(8)` for more information)
- Allow convergence modules to directly create PVCs just as they create SVCs

Either method creates a local reference to the PVC and allocates the resources necessary for the VC to function on the system. In either case, the convergence module's connection routine for the bind point to which the PVC is attached is called.

Note

You can create PVCs only on the PVC PPA. You cannot create a PVC on an SVC PPA.

When a convergence module sets up a PVC, the following events occur:

- A local reference to the VC is set up in the CMM.
- The ownership of the PVC is given to the convergence module.
- The VC is created and enabled in the device driver.

Once set up, the PVC remains in existence until explicitly torn down by the convergence module.

The ATM subsystem treats a PVC like an SVC, except that there is no signaling protocol associated with the PVC, and PVCs do not age. When a convergence module requests a PVC, it must specify all the VC service parameters as it does for an SVC. If the PVC is created successfully, this does not necessarily mean that the PVC exists. Even though a local reference for the PVC has been created, modules cannot check (due to the nature of PVCs) to determine if the PVC exists on the network.

Note

If you create a PVC before it is configured on the network, no invalid VCI errors are reported to the convergence module.

To release PVCs, use the `atm_cmm_release` function call. However, since PVCs require no signaling protocol exchange with the network, PVC release is synchronous. The return value from the release function call indicates success or failure of the teardown. The convergence module receives no other notifications.

7.6 Creating Signaling Virtual Circuits

Signaling VCs are well-known PVCs that signaling protocols use to communicate with the switches on the network. The switches and the end systems on an ATM network automatically set up signaling VCs. The ATM subsystem treats signaling VCs as ordinary PVCs and does not provide any special mechanisms for them. The signaling protocol module is responsible for managing the signaling VCs.

When signaling modules are initialized, they register with the CMM as both a signaling module and as a convergence module. The signaling

modules use the signaling module interface to field connection management requests from the CMM; they use the convergence module interface to send and receive signaling protocol packets on the network. Once registered as a convergence module, signaling modules must create all VCs required by their signaling protocol.

Once the signaling VCs are created, the CMM treats them like any other PVC. Like other convergence modules, the signaling module receives no notification whether a PVC exists on the network. Signaling modules receive notification of interfaces being brought up and interfaces going down. When an interface goes down, all signaling VCs on that interface are destroyed.

Module Management Interface

The Connection Management Module (CMM) provides an interface that allows any ATM module to exchange configuration and management information with application processes without the usual need for kernel modifications and new `/dev` nodes. The Module Management Interface (MMI) is the standard DIGITAL UNIX `ioctl` interface, but it allows applications to communicate with specific ATM modules directly through the CMM. See `ioctl(2)` for more information.

To exchange information with any ATM module, applications must open the `/dev/atm_cmm` device and issue `ioctl` system calls. The `ioctl` commands are executed either by the CMM directly or passed to a selected ATM module. The ATM module transfers all data associated with the `ioctl` commands to and from user space; the MMI always has a user context since it is executed as part of a system call.

New ATM modules can define their own `ioctl` commands independently since commands are routed only to a module selected by using an `ioctl` call. Modules can also define their own structures and protocols for exchanging data with applications since interpretation of the `ioctl` data argument is done by agreement between the module and its application, not by the CMM.

Applications cannot send the CMM any commands other than those commands for establishing MMI paths. Only applications that DIGITAL has written to manage the ATM subsystem can select the CMM as the target for commands. Also, a module can restrict the establishment of an MMI path through the use of key values that are known only to the module and applications that are allowed to communicate with it. This prevents applications from illegally communicating with ATM modules.

This chapter describes the following:

- How to create an MMI command path
- How to define new MMI `ioctl` commands
- The MMI calling conventions
- Device driver, signaling, and convergence module interfaces

8.1 Creating an MMI Path

To send `ioctl` commands to an ATM module, you must create a path through the CMM to the module. The following code fragment shows a routine that creates an MMI path:

```
#include <sys/atm.h>

make_mmi_path(char *name)
{
    int atm_fd;
    struct atm_mmi_path path;    ❶

    atm_fd = open("/dev/atm_cmm", O_RDWR);    ❷
    if(atm_fd == -1)
    {
        perror("open of /dev/atm_cmm");
        exit(errno);
    }
    path.name = name;
    path.length = strlen(name);
    path.key = 0x123456789abcdef;
    if(ioctl(atm_fd, GIOC_MMI_PATH, &path) == -1)    ❸
    {
        perror("GIOC_MMI_PATH");
        exit(errno);
    }

    return atm_fd;
}
```

❶ Declares the `atm_mmi_path` structure.

❷ Opens the `/dev/atm_cmm` device.

❸ Issues the `ioctl` call with the `GIOC_MMI_PATH` command. The argument to the `GIOC_MMI_PATH` command is the name of the module to which the path is to be established; each module must register with a unique name.

After the path is created, all subsequent `ioctl` commands not understood by the CMM are sent to the selected module.

The `/dev/atm_cmm` device has the following restrictions:

- You can use this device only for the management and administration of ATM modules, not for transferring data across the network.
- Only one application can have the device opened at any time. Multiple accesses to this interface are not permitted.
- Only applications with root permissions can open the device.

Since all ATM modules, including the CMM, share a common interface, an application can inadvertently or deliberately issue an incorrect or damaging `ioctl` call. To prevent this from occurring, applications can identify themselves to ATM modules when the MMI path is created so their privilege to contact the module can be confirmed. You are not required to implement this in your module, but should for added safety.

The application performs extra validation by passing a 64-bit key to the ATM module when the MMI path is created. The ATM module checks this key to verify that it is valid. Since only applications written specifically to communicate with certain modules know the key the module expects, it is more difficult for another module to create the MMI path to the ATM module.

Table 8–1 lists member names of the `atm_mmi_path` structure, with their associated data types, that modules might reference.

Table 8–1: The `atm_mmi_path` Structure Members

Member Name	Data Type
<code>*name</code>	<code>char</code>
<code>length</code>	<code>int</code>
<code>key</code>	<code>long</code>

The `name` member is a pointer to the name of the ATM module to which the MMI path is to be established. This name is a standard NULL-terminated ASCII string that must exactly match the string the module used when it registered with the CMM.

The `length` member is the length (in bytes) of the string pointed to by the `name` member. This length does not include the NULL termination.

The `key` member is a value that the application and the target ATM module agree to use to identify the application to the ATM module. Only target ATM modules use this member.

8.2 Defining New MMI `ioctl` Commands

All MMI `ioctl` commands are in the “g” `ioctl` group and are named `GIOC_command_name`. Each `ioctl` command group can contain up to 256 commands; each `ioctl` command encodes the group, the command within the group, the data transfer direction, and the size of the data to transfer. The GIOC group is divided into CMM-specific commands (with a range of 0 to 127, inclusive) and module commands (with a range of 128 to 255, inclusive). The CMM does not interpret commands in the module command

range. Also, modules are not passed commands in the CMM command range.

As a rule, the module and application program must agree on the command number and the format of the command data passed as the third argument to the ATM `ioctl` system calls for a module. You do this with a module specific header file in which the commands and structures are defined. This header file should include `sys/ioctl.h`, which contains the macros used to define `ioctl` commands.

To define a new `ioctl` command, you need the following pieces of information:

- The direction of any data transfer between the application and the ATM module (input from the application, output to the application, or both). This determines which macro to use for command definition (`_IOR`, `_IOW`, or `_IORW`).
- The definition of the data structures to be passed between the application and the ATM module as the argument to the command. This, along with the transfer direction information, controls how much data the kernel will move between user space and the kernel when the `ioctl` is processed.
- The number of the command (between 128 and 255).

Once you know this information, you can define a new `ioctl` command. For example, suppose an application needs to pass information contained in a single longword and receive information from the ATM module in the same word. You would define the command as follows:

```
#define GIOC_NEW_CMD  _IOWR('g',128,sizeof(long))
```

Note that the “g” must be in lowercase.

Using this example, the application executes the following call to send the new command to the ATM module:

```
long arg = some_value;
int atm_fd;

atm_fd = make_mmi_path("module name");
if(ioctl(atm_fd, GIOC_NEW_CMD, &arg) == -1)
{
    perror("GIOC_NEW_CMD");
    exit(errno);
}

printf("GIOC_NEW_CMD returned %ld", arg);
```

An `ioctl` system call can copy only fixed-size data directly addressed by the argument. If a command requires the exchange of variable size data,

the argument to the `ioctl` system call must be a fixed size structure that contains pointers to and lengths of variable size data in the application's memory space. The ATM module is responsible for copying data from or to this memory when processing the command; the kernel `ioctl` call copies the contents of the structure.

8.3 Using MMI Calling Conventions

All modules that provide an MMI routine must adhere to the following rules when handling arguments from the CMM:

- All MMI routines are called while in a system call context with no external locks held. This means that the MMI routines can block, if necessary, provided they do not hold simple locks while blocking.
- All MMI routines must return `ESUCCESS` to indicate successful completion of the operation or a DIGITAL UNIX error number to indicate the type of error that occurred.
- MMI modules can use the `retval` pointer to provide a nonzero return value to the calling program. This value is the return value from the `ioctl` when the MMI function returns `ESUCCESS` to the CMM. This follows the normal DIGITAL UNIX `ioctl` processing conventions.
- MMI routines are required to process only their own commands and one global command: the `GIOC_MMI_PATH` command. This command instructs the MMI function to verify that it is willing to accept the establishment of an MMI path from the CMM as requested by an application program. The argument to this command is a pointer to an `atm_mmi_path` structure that the MMI function is to evaluate.
- The MMI routine should verify that the key provided in the `atm_mmi_path` structure is correct, but might also verify the specified module name. This is redundant since the CMM already verifies the module name. If the key or module name provided is acceptable, the MMI function returns `ESUCCESS`, otherwise, it returns `ENXIO`.
- The MMI function is informed only when MMI paths are established, not when they are destroyed.

See Appendix A for information on the `xxx_mmi` function.

8.4 Using the Device Driver MMI

When an ATM device driver registers with the CMM, it provides the CMM with the address of the following management functions:

- The internal management function that the CMM uses to manage driver resources and VCs. Only the CMM uses this interface; it is not available through the MMI. This function is not allowed to block at any time.

- The `xxx_mmi` function. This function receives only MMI commands through the `ioctl` system call from an application that has been written specifically to manage the device driver.

The standard DIGITAL UNIX ATM interface defines no generic driver management through the MMI. If a device driver does not need to process MMI commands, it should set this argument in the `atm_cmm_register_dd` routine to NULL.

The device driver's MMI function is passed the following arguments when called:

- The device driver's module handle, which the device driver module provided the CMM at registration time
- The `ioctl` command and data value

The MMI function is permitted to block, if necessary, at any time and has access to the user context of the calling application.

8.5 Using the Signaling Module MMI

The signaling module MMI is similar to the convergence module and device driver MMIs. The signaling module is passed the following arguments when called:

- The signaling module handle, which the signaling module provided the CMM at registration time. The signaling module can use this value in any way it chooses. The CMM does not modify this value.
- The `ioctl` command and data value.

8.6 Using the Convergence Module MMI

When a convergence module registers with the CMM, it can supply the address of an MMI function to be called when an application creates an MMI path to the convergence module. If a convergence module does not provide an MMI management function, the module writer should register an address of NULL for the MMI management function when the convergence module registers with the CMM.

The convergence module MMI function is passed the following arguments when called:

- The convergence module's internal handle, which the convergence module provided to the CMM at registration time. This value is internal to the convergence module; the CMM does not modify this value.
- The `ioctl` command and data value.

Queuing Guidelines

The Connection Management Module (CMM) is essentially a switch that connects the various components of the DIGITAL UNIX ATM subsystem. The CMM provides no queuing mechanisms in either the transmit or receive data paths. This means that each component is free to implement a queuing policy that is appropriate for its protocol or hardware. This chapter discusses guidelines for ATM component developers to consider when designing queuing policies for a component.

9.1 Queuing in Device Drivers

Device driver writers must implement a queuing policy on the transmit path and might implement a queuing policy on the receive path. The following sections describe the characteristics of each.

9.1.1 Device Driver Transmit Queuing

Device drivers must do all queuing of outgoing data. Under normal conditions, convergence modules pass data to the device drivers as the data comes down from the protocol stacks. The device driver must either queue the data or reject it, but should not drop it unless an error occurs after the driver has accepted the data for transmission. There is no mechanism other than the flow-control mechanism for a driver to request more data from a convergence module.

Most ATM devices provide at least one hardware queue or ring in which a small number of packets or raw cells can be queued to the hardware. The size of this queue is usually limited by the hardware. Some adapters might be able to schedule the processing of cells from these queues based on quality of service (QOS) parameters associated with each queue to provide QOS-based servicing of virtual circuits (VCs). Simpler hardware generally implements only a single first-in, first-out (FIFO) queue. Because of the limited size of these queues, the drivers should provide a software queue that provides some buffering of data from the CMM to the hardware queues. The software queues permit the driver to better handle bursts of data without having to use flow-control techniques on the sender.

Device drivers can completely hide their queuing mechanisms and policies or advertise some or all of them to the CMM. For example, a device driver could implement the following queuing policies:

- A device driver with multiple hardware queues advertises only a single software queue to the CMM and decides how to queue each packet it receives from the CMM.
- A driver advertises all its hardware queues to the CMM and lets the CMM make the decisions about what VC is associated with each queue.
- A combination of the first two policies.

Regardless of the method implemented, device drivers should always provide a software queuing mechanism to handle conditions when the hardware queue is full in order to delay using flow-control techniques on the sender.

The driver advertises virtual queues, which can be implemented in any number of ways internally, to the CMM. The CMM handles only the virtual queues, viewing each queue as being able to support a single QOS and being serviced by the driver based on the QOS parameters that the CMM set.

In addition to its no queuing policy, the CMM does not enqueue data to the driver. When the CMM has data to transmit, it calls the driver's transmit routine. This routine must place the data passed to it on the correct queue. Thus, the transmit queuing policy is contained entirely within the device driver. This applies to both drivers that advertise single queues and those that advertise multiple queues.

The CMM assists drivers with multiple queues by assigning the VCs to the queues. In this case, the queue to which the CMM has assigned the VC is contained in the `atm_vc_services` structure referenced by the `atm_vc` structure. The driver uses this information to determine how to queue each packet of data.

If a device driver advertises only a single queue when it registers with the CMM, the CMM expects the device driver to provide its own internal scheduling of packet transmission.

If a device driver advertises more than one queue, the CMM tries to assign VCs to each queue in an effort to achieve the QOS requested for each VC. The CMM informs the driver of its intention to send data to a specific queue so the driver need not schedule the queue for servicing until notified by the CMM that the queue will be used. The CMM selects one queue for queuing best-effort available bit rate (ABR) traffic as VCs with this QOS are created. All ABR traffic is assigned to this queue. Queues for VCs that require other than an ABR QOS are chosen by the CMM as the VCs are created.

When the CMM needs to use a previously unused queue, it sends the driver an `ATM_DRVMGMT_SETQ` command through the driver's management interface. The argument to this command is a pointer to an `atm_vc_services` structure that contains all the information about the various QOS parameters for the queue. The driver might maintain a reference to this structure as long as the queue is in use by the CMM. The driver uses the information in this structure to schedule its servicing of its active queues. The driver must schedule the services of the queues so that the QOSs specified for all queues are met. The CMM also uses this command to change a queue's parameters, for example, when another VC is attached to the queue and the aggregate reserved bandwidth for the queue must be increased.

When the CMM closes the last VC associated with a queue, it notifies the device driver that the queue is no longer in use by sending it the `ATM_DRVMGMT_CLEARQ` command through the driver's management interface. The argument to this command is the number of the queue to clear; queues are numbered from 0 to the maximum number the driver supports minus 1. When the driver receives this command, it can stop servicing the queue because the CMM will not send any more data to that queue.

9.1.2 Device Driver Receive Queuing

Device drivers might supply a small amount of queuing on the receive path to reduce interrupt overhead, batching up several receive packets for processing on a single interrupt. However, driver writers should be careful not to introduce significant latency on incoming data unless they are capable of scheduling receive processing in a way that gives priority to constant bit rate (CBR) types of traffic.

9.2 Queuing in Convergence Modules

Like device drivers, convergence modules can implement any queuing mechanism on both the transmit and receive data paths. You can tailor the queuing mechanism implemented to the protocol's needs and the QOS requirements of the convergence module. Convergence modules are not required to queue in any direction, but doing so might improve performance by reducing data losses when there is congestion on the outgoing line.

9.2.1 Convergence Module Transmit Queuing

Since device driver writers must implement some sort of queuing on the transmit path, you can write convergence modules to leave all queuing of transmit data to device drivers. However, if a device driver queue becomes

full, convergence modules are responsible for deciding the disposition of any data that cannot be queued to the driver. If a driver cannot accept more data for transmission because of a full queue, the transmitting convergence module can simply elect to discard the data and try again later. However, this approach is not appropriate for all protocols, and may even cause performance problems for protocols that permit the arbitrary dropping of data. In such cases, the convergence modules must provide some queuing mechanism to hold the outgoing data until the driver is ready to accept more data for transmission (see Chapter 10 for an explanation of flow control in the ATM subsystem).

Since device driver queues are not visible to convergence modules, convergence modules must queue on a per-VC basis. At the convergence module interface, all flow control takes place on a per-VC basis, so convergence modules have to handle queuing for each VC (and can even elect to provide queuing on only certain connections and discard data on others). When a convergence module receives an `ATM_CAUSE_QWARN` or `ATM_CAUSE_QFULL` return from the `atm_cmm_send` call, this means that the driver's queue is almost full or full, respectively, and it cannot accept the data. At this point, the VC is considered *flow controlled*. The convergence module must either queue subsequent transmit data or discard the data until it receives a flow-control notification from the CMM.

The convergence module's implementation must determine the size of the module's per-VC queues. This implementation depends on the amount of data expected to flow on the VC and the ability of the protocol to retransmit data. When the device driver is ready to accept more data for transmission, it notifies the CMM, which in turn notifies the convergence module. When the convergence module receives the notification, it can start sending queued data until its queue is either drained or until it receives another indication from the CMM that the driver's queue is full. Once the queue has been cleared, all further data can be sent directly to the CMM for transmission, avoiding the overhead of queuing and dequeuing.

9.2.2 Convergence Module Receive Queuing

Convergence modules are generally passed incoming data while the processor is still running off the device driver's interrupt stack (from the receive interrupt). This means the convergence module receive routines are running at a high priority and should process the data and return to the CMM as quickly as possible. Processing of receive data in this fashion is allowed so that the CMM receive path introduces no additional latency that may be unacceptable for CBR traffic.

The DIGITAL UNIX ATM subsystem attempts to deliver the data from the adapter to the convergence module as quickly as possible, with a minimum

of extra processing and no additional latency. That way, convergence modules that handle CBR traffic are passed the incoming data as quickly as possible without interference from the ATM infrastructure or from the scheduling latencies of the operating system. Therefore, convergence modules must implement a receive queuing mechanism that is both appropriate to their protocol and QOS needs and that makes the modules cooperate in a multiprotocol, multi-QOS environment. For example, a convergence module that handles ABR traffic should not hold the processor for longer than necessary just because it can and the module writer might want that module to perform well at the expense of other protocols.

In general, convergence modules should queue incoming data as soon as they can and schedule a kernel thread to process the data at a later time. Of course, modules that process CBR traffic should not queue data, but should do as much processing as is appropriate. Too much processing on the interrupt stack, however, could lead to live lock conditions on the processor.

Convergence modules cannot use flow control on device drivers. The only methods available to enforce flow control on the sender is either protocol-specific flow control or ATM flow control. Device drivers will use ATM flow control when they start running out of receive buffer space.

10

Flow Control

The DIGITAL UNIX ATM subsystem uses the following types of flow control:

- **Hardware flow control** — The ATM network uses this to manage congestion.
- **Software flow control** — The DIGITAL UNIX ATM subsystem uses this to control the movement of data between modules.

Although these methods of flow control are different, they do interact, especially in the transmission of data on the network. You must design and implement each device driver and convergence module to use flow control properly, thus ensuring the proper operation of the entire subsystem.

10.1 Hardware Flow Control

The ATM network and adapter use hardware flow control to control congestion within the network. This allows the network to prevent the sender from transmitting more data than the network can handle. In the UNI 3.0/3.1 environment, there is no standard for ATM flow control, although many vendors implement their own methods of flow control. Since these flow-control methods are confined to the ATM adapter and since the DIGITAL UNIX ATM subsystem accommodates as many types of adapters as possible, all methods of flow control are accommodated as long as they are isolated to the adapter and its device driver. The ATM subsystem also supports the use of both proprietary and standard methods of hardware flow control.

To enable hardware flow control, you must send a device driver the `ATM_DRVMGMT_FC` command through the driver's management function. The argument to this command indicates whether to enable flow control (and the type to enable) or to disable all flow control. Drivers must be initialized with flow control disabled. Currently, the CMM recognizes the following types of hardware flow control:

- **Vendor-specific flow control**

Requires that the adapter and the switch implement the same type of hardware flow control. This command is sent as the result of the `atmconfig +vfc` command being issued by an operator (the system is

unable to determine which types of vendor-specific protocols are usable since this requires knowledge of the protocols that both the adapter and the switch implement).

- ATM Forum standard flow control

Currently not supported. Generally, the CMM uses the fields in the switch and driver Management Information Bases (MIBs) to detect flow control, and turns on this flow control without any user interaction. However, system administrators can also enable this form of flow control by running the `atmconfig +sfc` command.

To disable flow control, a system administrator uses either the `atmconfig -sfc` or `atmconfig -vfc` command, depending on the type of flow control currently enabled.

10.2 Software Flow Control

If you do not use hardware flow control, the driver might be passed data from convergence modules faster than it can transmit the data on the interface. In this case, the driver must be able to inform the sending convergence modules that its queues are full and that no data should be sent until further notification. Rather than having the driver perform an explicit notification (which involves extra function call overhead) to do this, the driver must return a value from its transmit function indicating that its queue is full (or is filling). The driver should then consider the queue flow controlled and must send a notification to the CMM when the queue is ready to accept more data.

In its implementation, the CMM records most information associated with flow control. Since the driver does all software flow control on a per-queue basis, the CMM must translate this into information that can be used on a per-VC basis. The CMM maintains a list of which VCs are assigned to which driver queues; the CMM converts driver queue enable notifications to per-VC enable notifications for convergence modules.

Drivers must maintain the following values on their queues: a high-water mark and a low-water mark. Drivers use these marks to control the value returned by the transmit function and the sending of flow-control notifications to the CMM.

10.2.1 High-Water Mark

The high-water mark controls when the driver starts returning warnings (`ATM_CAUSE_QWARN`) from the transmit call. These warnings are used by convergence modules that might try to start queuing before the driver can no longer accept data. The use of warnings also allows convergence

modules the chance to use flow control on the upper-layer protocol modules, if possible, before the driver actually starts rejecting data. The driver should start returning the queue warning when the number of queued messages passes its high-water mark. The driver writer must choose a high-water mark that is appropriate for the driver's queuing policy and adapter characteristics.

When a driver returns a queue warning, it must have accepted the data for transmission and queued the data. The warning returned does not indicate that the driver cannot accept the data, only that the queue is close to being full.

When a driver's queue is full, the driver must not discard data passed in the transmit function. Instead, the driver must return `ATM_CAUSE_QFULL` from the transmit function so the convergence module can decide if the data is to be dropped or queued within the convergence module. The queue full return indicates that the driver cannot accept the data for transmission because its queue is full, not for any other reason, such as insufficient memory. Drivers can continue to receive and reject packets for transmission even after the queue fill return, since more than one convergence module could be queuing to the queue.

10.2.2 Low-Water Mark

When the number of messages on a driver's transmit queue drops below the low-water mark, the driver sends an `ATM_DE_STARTQ` notification to the CMM. The driver must send this notice if it has returned any queue warning or full conditions from its transmit function since the last restart notification was sent. This implies that drivers maintain state information about previous returns and notifications. Driver writers can provide extra restart notifications (the CMM properly handles redundant notification), but should not provide too few notifications. The driver notifies restarts per queue only. The CMM converts these to per-VC restarts and sends those notifications to convergence modules. Therefore, drivers need to maintain only the per-queue state, not the per-VC state, for flow control.

10.3 Convergence Module Flow Control

Convergence modules are not required to maintain any type of transmit queue since ATM device drivers must provide some amount of output queuing. However, convergence modules should provide transmit queues if data not accepted for transmission by a driver is to be saved for retransmission. If it is acceptable to discard data, convergence modules can ignore queuing and flow-control features all together.

Device drivers generate flow control on a per-queue basis; convergence modules handle flow control on a per-VC basis. The reasons for this are as follows:

- Information concerning driver queuing policies and CMM assignment of VCs to driver queues is not available to convergence modules.
- The CMM uses flow control on the sender synchronously so that only those VCs trying to send data to a flow-controlled queue actually receive flow-control indications. Those VCs that do not send any data to the affected queue while the flow-control condition exists never receive a flow-control indication and are unaffected by the flow-control condition.
- Only those VCs that have been flow controlled are notified when they can resume transmission of data.

If a convergence module implements transmit queuing, it must be prepared to handle software flow control from the CMM. Convergence modules are given flow-control information in the return from the CMM send routine and by an exception notification. The convergence module should use this information to control its queuing and sending of data.

Convergence modules can bypass internal queues for efficiency as long as data is flowing freely to the CMM. A convergence module needs to use transmit queues only when it is informed that the device driver's queue is filling up or is full. Once the queues have drained, the queues can be bypassed again. This type of implementation requires some extra state to be maintained in the convergence module. However, this implementation could increase the speed of the nonqueued transmit path.

When a convergence module calls `atm_cmm_send`, it receives an `atm_err_t` return value, indicating the disposition of the data by the CMM and device driver. If the device driver accepts the data for transmission, `atm_cmm_send` returns `ATM_CAUSE_GOOD`. In this case, the driver has queued the data for transmission and the convergence module must destroy any references to the data; the driver frees the mbufs associated with the data.

When a device driver's transmit queue fills up beyond the high-water mark, the driver starts returning `ATM_CAUSE_QWARN` to the CMM, which returns this value to the convergence module. When a convergence module gets an `ATM_CAUSE_QWARN` back from `atm_cmm_send`, this means that the data was accepted and queued by the driver, but that the convergence module should not send any more data. The driver's queue could fill up and it would have to start rejecting data. In this case, the convergence module should either start queuing outgoing data or start dropping subsequent packets until it receives notification that the driver can accept data again.

The convergence module might also notify the upper-layer protocols (if the flow can be controlled) and continue to send data to the driver until the upper layers stop sending. A queue warning might not be returned before a queue is completely full and the driver can no longer accept data. This is true in cases in which multiple VCs are feeding a single queue. The queue warning is intended to give the convergence module warning of an impending queue full condition so that it can take any appropriate action *before* the driver stops accepting data.

If a device driver's queue becomes completely full and the driver is unable to queue any additional data, `ATM_CAUSE_QFULL` is returned from `atm_cmm_send`. In this case, the driver did not accept and queue the data and the convergence module must consider the VC blocked and unable to transmit any more data. The module must either queue the data internally or discard it. Any further attempts to transmit will probably return a queue full condition.

For those cases where multiple VCs feed a single queue and the convergence module can control the upper-layer protocols, the convergence module should implement a small transmit queue to handle any data that is sent by the upper layers between the flow-control notification and the actual cessation of data transmission. A small transmit queue ensures sufficient storage space to prevent data loss if a driver queue suddenly becomes full due to the accumulation of data from another VC.

When a queue warning or a queue full condition is reported to a convergence module, the VC that received the condition is considered flow controlled; no more data should be sent on the VC. When the driver queue drains below its low-water mark, the driver notifies the CMM that it is ready to start accepting data on that queue. The CMM then sends an `ATM_CME_START_VC` notification to each convergence module, specifying that the affected VCs can resume data transmission. One notification is sent for every VC that receives a flow-control indication from one or more `atm_cmm_send` calls.

When a convergence module receives the `ATM_CME_START_VC` exception, it should arrange to start sending data from its queues down to the driver and to remove flow control from the upper-layer protocols, if necessary. Note that convergence modules should not drain their queues as part of the exception function call on which the `ATM_CME_START_VC` exception is received as this could be on an interrupt stack. Instead, convergence modules should schedule a kernel thread or timeout to run and process the queued data.

Tuning the ATM Subsystem

Table 11–1 lists the configuration attributes for tuning the ATM subsystem. Use the `sysconfig -q atm` command to query the default values. See *System Configuration and Tuning* for more information on tuning subsystem operations.

Table 11–1: ATM Subsystem Configuration Attributes

Attribute	Description
<code>resvtimeo</code>	The number of seconds after which unused reserved resources are revoked via an <code>ATM_CME_RESV_EXPIRE</code> exception notification. The default is 30 seconds.
<code>sysresvlim</code>	The percentage of each driver's total interface bandwidth that is available to constant bit rate (CBR) and pacing circuits in the backward or incoming and forward or outgoing directions. The value is specified as an integer (0-100). Use the <code>atmconfig</code> command to set this limit on a per-driver basis.
<code>fsysresvlim</code>	The percentage of each driver's total interface bandwidth that is available to constant bit rate (CBR) and pacing circuits in the forward or outgoing direction. The value is specified as an integer (0-100). Use the <code>atmconfig</code> command to set this limit on a per-driver basis.
<code>bsysresvlim</code>	The percentage of each driver's total interface bandwidth that is available to constant bit rate (CBR) and pacing circuits in the backward or incoming direction. The value is specified as an integer (0-100). Use the <code>atmconfig</code> command to set this limit on a per-driver basis.

Table 11–2 lists the configuration attribute for tuning the UNI3X subsystem. Use the `sysconfig -q uni3x` command to query the default values. See *System Configuration and Tuning* for more information on tuning subsystem operations.

Table 11–2: UNI3X Subsystem Configuration Attributes

Attribute	Description
timer_resolution	The number of ticks per second of the signaling module timers. The default is 10 ticks per second, or 100-millisecond resolution. If you increase this value, you achieve better precision but with greater processor overhead.

Table 11–3 lists the configuration attribute for tuning the LANE subsystem. Use the `sysconfig -q lane` command to query the default values. See *System Configuration and Tuning* for more information on tuning subsystem operations.

Table 11–3: LANE Subsystem Configuration Attributes

Attribute	Description
nelan	The maximum number of emulated LANs that you can configure on the system. The default is 4. The maximum number is determined by the number of ESIs available on the system.

A

CMM Routines

This appendix contains a description of each of the routines described in this guide, in reference-page format. The routines are in alphabetical order.

atm_cmm_accept

NAME

`atm_cmm_accept` – Accepts a previously deferred call and informs the CMM that the call should be accepted

SYNOPSIS

```
atm_error_t atm_cmm_accept(  
    atm_addr_p addr,  
    atm_vc_services_t give);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure for the call being accepted. This is the same pointer passed into the convergence module's `xxx_connect` routine when the call first came in.

give This argument is not currently used.

DESCRIPTION

The `atm_cmm_accept` routine is a convergence module interface that accepts a previously deferred incoming call, notifies the CMM, and sets the circuit resource parameters. The incoming call was deferred when a convergence module returned `ATM_CAUSE_DEFER` from its `xxx_connect` routine.

Typically, the determination is done within a set period of time, as defined by the signaling protocol, before the calling party times out on the call request.

The connection is not established and ready for use until the convergence module receives the appropriate exception notifications.

RETURN VALUES

If connection setup is proceeding, `atm_cmm_accept` returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number that indicates the reason the call failed. If the call fails, the convergence module should remove any reference to the call.

`atm_cmm_accept`

RELATED INFORMATION

`xxx_connect`

Section 7.2 for information on accepting connections

`atm_cmm_activate_con`

NAME

`atm_cmm_activate_con` – Informs the CMM that a connection is ready to carry data

SYNOPSIS

```
atm_error_t atm_cmm_activate_con(  
    atm_sig_handle_t sm,  
    atm_addr_t *addr);
```

ARGUMENTS

- sm* Specifies the signaling module handle that the CMM returned in the registration call.
- addr* Specifies a pointer to an `atm_addr` structure that contains all the parameters of the call. This pointer must point to the same `atm_addr` structure that was passed in either the `atm_cmm_new_call` function call or in the `xxx_setup` and `xxx_add` function calls since this pointer is the handle that refers to the specific connection.

DESCRIPTION

The `atm_cmm_activate_con` routine is a signaling module interface. Signaling modules call this routine when they receive an indication from the network that the circuit is connected. The routine notifies the CMM that the connection is ready to carry data. The CMM enables the VC locally, and passes the notification to the device driver and protocol convergence modules.

Note

Do not make this call before circuit connection.

RETURN VALUES

If the indicated connection is found on the system no matter what the result of the operation, `atm_cmm_activate_con` returns

`atm_cmm_activate_con`

ATM_CAUSE_GOOD. If the indicated connection is not found, an error value is returned indicating that the signaling module should release the connection since the CMM has no reference to it.

RELATED INFORMATION

`atm_cmm_con_release`, `xxx_setup`, `xxx_add`

Section 7.4 for information on releasing connections

`atm_cmm_add`

NAME

`atm_cmm_add` – Adds an endpoint to an existing point-to-multipoint connection

SYNOPSIS

```
atm_err_t atm_cmm_add(  
    atm_cvg_handle_t cm,  
    atm_addr_t *addr,  
    atm_uni_call_ie_p ei,  
    atm_vc_t *vc);
```

ARGUMENTS

- | | |
|-------------|---|
| <i>cm</i> | Specifies a value returned to the convergence module by the registration function call. This uniquely identifies the convergence module making the request. |
| <i>addr</i> | Specifies a pointer to a properly set <code>atm_addr</code> structure that specifies the address of the endpoint to be added to the connection. |
| <i>ei</i> | Specifies a pointer to an array of <code>atm_uni_call_ie</code> structures that has been initialized with information to be used in placing the additional call. This argument must be NULL if no optional IEs are specified. |
| <i>vc</i> | Specifies a pointer to the <code>atm_vc</code> structure of the VC to which the endpoint is to be added. The VC must have been created with the <code>ATM_CT_PTM</code> flag set. |

DESCRIPTION

The `atm_cmm_add` routine is a convergence module interface that adds endpoints to an existing point-to-multipoint connection. You can add endpoints at any time only to connections that were created with the `ATM_CT_PTM` flag set.

`atm_cmm_add`

EXAMPLE

See Example B-2 for a code example of the `atm_cmm_add` routine.

RETURN VALUES

If the endpoint add is proceeding, `atm_cmm_add` returns `ATM_CAUSE_GOOD`. The convergence module receives the `ATM_CME_CALL_FAILED` or the `ATM_CME_EP_ACTIVE` exception indications to report the progress of the call. `ATM_CME_EP_ACTIVE` indicates that the call completed and the connection to the endpoint is active and can transport data. These indications do not reflect the state of any other endpoint associated with the VC. If the endpoint add cannot proceed, an ATM error number is returned.

RELATED INFORMATION

`atm_cmm_drop`

Chapter 7 for information on connections

`atm_cmm_adi_set_cause`

NAME

`atm_cmm_adi_set_cause` – Logs a network-visible VC or endpoint condition

SYNOPSIS

```
atm_err_t atm_cmm_adi_set_cause(
    atm_drv_handle_t driver,
    atm_vc_p vp,
    atm_addr_p addr,
    char *reason,
    atm_error_t cause,
    atm_location_t location,
    unsigned char diag_length,
    unsigned char *diag);
```

ARGUMENTS

<i>driver</i>	Specifies the interface on which the condition occurred. This is the driver handle that the CMM assigned at driver registration time.
<i>vp</i>	Specifies a pointer to the VC on which the condition occurred. If the error applies to one endpoint, set the value to NULL and set the <i>addr</i> value to non-NULL.
<i>addr</i>	Specifies a pointer to the <code>atm_addr</code> structure for the endpoint on which the error occurred. If the error applies to all endpoints on the VC (global), set the value to NULL and set the <i>vp</i> value to non-NULL.
<i>reason</i>	Specifies a NULL-terminated character string that contains descriptive text for the error.
<i>cause</i>	Specifies the <code>atm_error_t</code> value that describes the condition.

atm_cmm_adi_set_cause

<i>location</i>	Specifies an <code>atm_location_t</code> value that indicates the location in the network where the error occurred.
<i>diag_length</i>	Specifies the length (in bytes) of the data in the <i>diag</i> argument.
<i>diag</i>	Specifies a string of bytes that contains diagnostic information about the error. This data might be passed to the network in the <i>diag</i> field of a Cause IE.

DESCRIPTION

The `atm_cmm_adi_set_cause` routine is a device driver interface that records an error or normal condition associated with a VC or endpoint. The condition is stored as an `atm_cause_info` structure together with the VC. Convergence and signaling modules can retrieve this information by using the `atm_cmm_next_cause` routine. Users can display this information by using the `atmconfig` utility.

Errors or events stored with the `atm_cmm_adi_set_cause` routine are visible to the network and to the other end of the connection. When a signaling module releases a connection or drops an endpoint, the module extracts the most recent cause from a VC or endpoint and creates a Cause IE.

RETURN VALUES

If the cause is recorded successfully, `atm_cmm_adi_set_cause` returns `ATM_CAUSE_GOOD`. If the driver handle, *vp* argument, or *addr* argument is invalid, `atm_cmm_adi_set_cause` returns `ATM_CAUSE_BARG`.

RELATED INFORMATION

`atm_cmm_adi_set_log`, `atm_cmm_next_cause`, `atm_cmm_set_cause`,
`atm_cmm_smi_set_cause`

Section 3.10.7 for information on cause information

`atm_cmm_adi_set_log`

NAME

`atm_cmm_adi_set_log` – Logs a VC or endpoint condition

SYNOPSIS

```
atm_err_t atm_cmm_adi_set_log(  
    atm_drv_handle_t driver,  
    atm_vc_p vp,  
    atm_addr_p addr,  
    char *reason,  
    atm_error_t cause,  
    atm_location_t location,  
    unsigned char diag_length,  
    unsigned char *diag);
```

ARGUMENTS

<i>driver</i>	Specifies the interface on which the condition occurred. This is the driver handle that the CMM assigned at driver registration time.
<i>vp</i>	Specifies a pointer to the VC on which the condition occurred. If the error applies to one endpoint, set the value to NULL and set the <i>addr</i> value to non-NULL.
<i>addr</i>	Specifies a pointer to the <code>atm_addr</code> structure for the endpoint on which the error occurred. If the error applies to all endpoints on the VC (global), set the value to NULL and set the <i>vp</i> value to non-NULL.
<i>reason</i>	Specifies a NULL-terminated character string that contains descriptive text for the error.
<i>cause</i>	Specifies the <code>atm_error_t</code> value that describes the condition.
<i>location</i>	Specifies the location in the network where the error occurred.

`atm_cmm_adi_set_log`

<i>diag_length</i>	Specifies the length (in bytes) of the data in the <i>diag</i> argument.
<i>diag</i>	Specifies a string of bytes that contains diagnostic information about the error. This data might be passed to the network in the <i>diag</i> field of a Cause IE.

DESCRIPTION

The `atm_cmm_adi_set_log` routine is a device driver interface that records an error or normal condition associated with a VC or endpoint. The condition is stored as an `atm_cause_info` structure together with the VC. Users can display this information by using the `atmconfig` utility.

Errors or events stored with the `atm_cmm_adi_set_log` routine are logged on the local system only, and are not available when a signaling module generates a Cause IE. Logging provides information about VC or endpoint activity for a system or network administrator to view on the local system.

RETURN VALUES

If the cause is recorded successfully, `atm_cmm_adi_set_log` returns `ATM_CAUSE_GOOD`. If the driver `handle`, `vp` argument, or `addr` argument is invalid, the routine returns `ATM_CAUSE_BARG`.

RELATED INFORMATION

`atm_cmm_adi_set_cause`, `atm_cmm_next_cause`, `atm_cmm_set_log`,
`atm_cmm_smi_set_log`

Section 3.10.7 for information on cause information

`atm_cmm_alloc_addr`

NAME

`atm_cmm_alloc_addr` – Allocates memory for and initializes the `atm_addr` structure

SYNOPSIS

```
atm_addr_p atm_cmm_alloc_addr( void );
```

DESCRIPTION

The `atm_cmm_alloc_addr` routine is a signaling and convergence module interface that allocates memory for the `atm_addr` structure and initializes the structure members. One `atm_addr` structure is allocated by a convergence module for each endpoint it calls and by a signaling module for each endpoint that calls the local host.

Usually, the CMM frees all ATM address structures associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails), it may be necessary for the allocating module to free memory it has allocated. In these cases, modules can call the `atm_cmm_free_addr` routine with the value returned from `atm_cmm_alloc_addr` to free memory.

EXAMPLE

See Section B.1 for a code example of the `atm_cmm_alloc_addr` routine.

RETURN VALUES

If storage cannot be allocated, a NULL is returned; otherwise, a pointer to the allocated memory is returned.

RELATED INFORMATION

`atm_cmm_free_addr`

Section B.1 for information on connections

`atm_cmm_alloc_ie`

NAME

`atm_cmm_alloc_ie` – Allocates memory for and initializes the `atm_uni_call_ie` structure

SYNOPSIS

```
atm_uni_call_ie_p atm_cmm_alloc_ie(  
    int num_ie);
```

ARGUMENTS

`num_ie` Specifies the number of IE structures to allocate.

DESCRIPTION

The `atm_cmm_alloc_ie` routine is a convergence module interface that allocates memory for the `atm_uni_call_ie` structure and initializes the structure members.

Typically, the convergence protocol module frees all UNI signaling structures associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails) the allocating module (either the convergence or signaling module) might have to free memory it has allocated. Convergence modules should not free memory for this structure if the `atm_cmm_connect` or related call succeeds because the CMM will free the memory.

EXAMPLE

See Section B.1, Section B.2, and Section B.3 for code examples of the `atm_cmm_alloc_ie` routine.

RETURN VALUES

If storage cannot be allocated, a NULL is returned; otherwise, a pointer to the allocated memory is returned.

`atm_cmm_alloc_ie`

RELATED INFORMATION

`atm_cmm_free_ie`

Chapter 7 for information on connections

`atm_cmm_alloc_services`

NAME

`atm_cmm_alloc_services` – Allocates memory for and initializes an `atm_vc_services` structure

SYNOPSIS

```
atm_vc_services_p atm_cmm_alloc_services( void );
```

DESCRIPTION

The `atm_cmm_alloc_services` routine is a signaling and convergence module interface that allocates memory for the `atm_vc_services` structure and initializes the structure members to default values for best-effort available bit rate (ABR) service. Once allocated, other ATM modules can change the `atm_vc_services` structure as needed.

Typically, the CMM frees the `atm_vc_services` structure associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails), the allocating module might need to free memory it has allocated. In these cases, modules call the `atm_cmm_free_services` routine with the value returned from `atm_cmm_alloc_services` to free memory.

EXAMPLE

See Section B.1 for a code example of the `atm_cmm_alloc_services` routine.

RETURN VALUES

If storage cannot be allocated, a NULL is returned; otherwise, a pointer to the allocated memory is returned.

RELATED INFORMATION

`atm_cmm_free_services`

Chapter 7 for information on connections

atm_cmm_bind_info

NAME

atm_cmm_bind_info – Queries parameters from a bind

SYNOPSIS

```
unsigned long atm_cmm_bind_info(  
    atm_bind_handle_t handle,  
    atm_bind_info_t it);
```

ARGUMENTS

- handle* Specifies a pointer to a CMM bind handle of a currently active AESA binding. Information is retrieved from the PPA associated with the bind point.
- it* Specifies the type of information that is being queried. The value returned depends on the object type of the queried information. The following values may be used to query the indicated information:

Value	Meaning
ATM_BIND_INFO_HANDLE	Returns the convergence module's bind handle, which the convergence module supplied at the time it created the AESA bind point.
ATM_BIND_INFO_ID	Returns a unique global value for referencing the bind point. Each bind point gets assigned a unique ID (which is different from the bind handle). An application program can use this value when specifying a bind point (atmconfig uses these values to reference bind points). The returned value is 32-bits wide.
ATM_BIND_INFO_PPA	Returns a pointer to the PPA associated with the AESA bind point.
ATM_BIND_INFO_SELECTOR	Returns the selector value assigned to the bind point. Primarily, a convergence module uses this to request that the CMM assign an unused selector value to a bind point. The value returned is between 0 and 255 for SVC bindings, and is 31-bits wide for PVC bindings.
ATM_PPA_ALLOCGRANE	Returns a type of atm_bw_granularity_p, which reflects the underlying interface's units of bandwidth allocation, or its allocation granularity.

atm_cmm_bind_info

Value	Meaning
ATM_PPA_ALLOCLIMIT	Returns a type of <code>atm_bw_granularity_p</code> , which reflects the underlying interface's per-VC bit rate limits, expressed in allocation granularity units.
ATM_PPA_AVAILRES	Returns a type of <code>atm_services_granes_p</code> , which reflects the amount of bandwidth (in granularity units) currently available for new CBR and pacing circuits over the underlying interface.
ATM_PPA_BRESVLIM	Returns the user-configurable limit, expressed as a percentage of backward bandwidth, on CBR circuits.
ATM_PPA_ESI	Returns the ESI that the signaling module supplied when the PPA was created.
ATM_PPA_ESIID	Returns the unique ESI identifier associated with the PPA's ESI.
ATM_PPA_ESILEN	Returns the length of the ESI that the signaling module supplied when the PPA was created.
ATM_PPA_ESIPID	Returns the ESI identifier of the PPA's parent ESI.
ATM_PPA_FRESVLIM	Returns the user-configurable limit, expressed as a percentage of forward bandwidth, on CBR and pacing circuits.
ATM_PPA_INFO_BURST_AVAIL	Returns the available burst cell rate in cells-per-second. This is the amount of burst cell rate that is available to be reserved by a convergence module.
ATM_PPA_INFO_BURST_MAX	Returns the maximum burst cell rate supported by the interface in cells-per-second.
ATM_PPA_INFO_CAPABILITIES	Returns the driver's capabilities for that interface.
ATM_PPA_INFO_DID	Returns a unique global value that can be used to reference the driver for the underlying interface.
ATM_PPA_INFO_DNAME	Returns the name of the driver for the underlying interface.
ATM_PPA_INFO_DUNIT	Returns the driver's unit number for the underlying interface.
ATM_PPA_INFO_FC	Returns a nonzero value if hardware flow control is currently enabled on the interface.
ATM_PPA_INFO_HARD_MTU	Returns the largest PDU (in bytes) that the interface supports. This is valid only for interfaces that support AAL5 or AAL3/4 SAR functions in hardware.
ATM_PPA_INFO_HI_VCI	Returns the highest value that may be used for a VCI on the interface.
ATM_PPA_INFO_HI_VPI	Returns the highest value that may be used for a VPI on the interface.

atm_cmm_bind_info

Value	Meaning
ATM_PPA_INFO_ID	Returns a unique global value for referencing the PPA. Each PPA gets assigned a unique ID when it is created. An application program can use this value to specify a bind point (<code>atmconfig</code> uses these values to reference bind points). The returned value is 32-bits wide.
ATM_PPA_INFO_MAX_VCI	Returns the maximum number of VCIs that the interface supports.
ATM_PPA_INFO_MAX_VPI	Returns the maximum number of VPIs that the interface supports.
ATM_PPA_INFO_MEDIA	Returns a value that indicates the type of physical media to which the interface is connected. The return value is of the type <code>atm_media_type_t</code> .
ATM_PPA_INFO_PEAK_AVAIL	Returns the available peak cell rate (in cells-per-second). This is the amount of peak cell rate that is available to be reserved by a convergence module.
ATM_PPA_INFO_PEAK_MAX	Returns the maximum peak cell rate (in cells-per-second) that the interface supports.
ATM_PPA_INFO_QUEUES	Returns the number of scheduling queues that the driver has made visible to the CMM.
ATM_PPA_INFO_SNAME	Returns the name of the signaling module that created the PPA.
ATM_PPA_INFO_SUST_AVAIL	Returns the available sustainable cell rate (in cells-per-second). This is the amount of sustainable cell rate that is available for a convergence module to reserve.
ATM_PPA_INFO_SUST_MAX	Returns the maximum sustainable cell rate (in cells-per-second) that the interface supports.
ATM_PPA_INFO_TOTAL_VC	Returns the maximum number of VCs that can be opened on the interface at any given time.
ATM_PPA_INFO_TYPE	Returns a value that indicates the underlying interface type. The returned value is of the type <code>atm_interface_t</code> .
ATM_PPA_INFO_UNI	Returns the UNI type associated with the PPA, which the signaling module supplied at the time the PPA was created. The return value is of the type <code>atm_uni_type_t</code> .
ATM_PPA_INFO_VC_LEFT	Returns the number of unopened VCs on the interface at the time of the call. This is the total number of VCs (the value returned by <code>ATM_PPA_INFO_TOTAL_VC</code>) minus the number of VCs currently opened.
ATM_PPA_MAX_VC_BBW	Returns the user-configurable maximum per-VC backward (incoming) bit rate (in allocation granularity units) permitted over the underlying interface.

atm_cmm_bind_info

Value	Meaning
ATM_PPA_MAX_VC_FBW	Returns the user-configurable maximum per-VC forward (outgoing) bit rate (in allocation granularity units) permitted over the underlying interface.
ATM_PPA_MAXRES	Returns a type of <code>atm_services_granes_p</code> , which reflects the maximum bandwidths available for CBR and pacing circuits over the underlying interface. These bandwidths (expressed in granularity units) are a function of the interface limits and the user-configurable limits on the percentage of bandwidth available to CBR and pacing circuits.
ATM_PPA_NUM_VCI	Returns the current number of VCIs configured on the underlying interface.
ATM_PPA_NUM_VPI	Returns the current number of VPIs configured on the underlying interface.
ATM_PPA_PEAK_CELLRATE	Returns the peak PDU bit rate (in cells-per-second) for the underlying interface.
ATM_PPA_UNUSEDRES_BACK	Returns the amount of backward (incoming) bandwidth (in cells-per-second) currently reserved for, but not yet applied to, CBR circuits.
ATM_PPA_UNUSEDRES_FWD	Returns the amount of forward (outgoing) bandwidth (in cells-per-second) currently reserved for, but not yet applied to, CBR and pacing circuits.

DESCRIPTION

The `atm_cmm_bind_info` interface is a convergence module interface that queries the CMM for current information about an interface's physical capabilities (such as its bit rates) and the amount of specific resources left available on an interface (the remaining nonreserved sustainable bit rate). The CMM and device drivers keep track of this information.

Since convergence modules normally deal with either PPAs or AESA bindings, the CMM allows convergence modules to query the information from the underlying interface by specifying the bind directly. Thus, the convergence module requires no specific knowledge of any device driver or device driver-related structures.

RETURN VALUES

If an invalid query is made, `atm_cmm_bind_info` returns a value of 0.

`atm_cmm_bind_info`

RELATED INFORMATION

`atm_cmm_del_ppa`, `atm_cmm_new_ppa`, `atm_cmm_ppa_bind`,
`atm_cmm_ppa_info`

Section 4.4.1.15 for information on driver capabilities

`atm_cmm_con_deleted`

NAME

`atm_cmm_con_deleted` – Notifies the CMM that a connection does not exist

SYNOPSIS

```
atm_err_t atm_cmm_con_deleted(  
    atm_sig_handle_t sm,  
    atm_addr_p addr);
```

ARGUMENTS

- sm* Specifies the signaling module handle that the CMM returned in the registration call.
- addr* Specifies a pointer to an `atm_addr` structure that contains all the parameters of the call. This pointer must point to the same `atm_addr` structure that was passed in either the `atm_cmm_new_call` function call or in the `xxx_setup` and `xxx_add` function calls since this pointer is the handle that refers to the specific connection.

DESCRIPTION

The `atm_cmm_con_deleted` interface is a signaling module interface. A signaling module calls this routine when the module receives confirmation of a connection's release from the switch. The routine notifies the CMM that the connection no longer exists.

The CMM holds the resources associated with the connection for a brief period of time, then releases them, giving all incoming queues time to clear. The CMM also notifies convergence modules that the connection has been released.

This routine notifies the CMM of both the deletion of point-to-point connections and of the deletion of parties in point-to-multipoint connections.

`atm_cmm_con_deleted`

RETURN VALUES

If the endpoint referenced is invalid, the `atm_cmm_con_deleted` routine returns `ATM_CAUSE_EIR` to indicate that it has no previous entry for the connection; otherwise, it returns `ATM_CAUSE_GOOD`.

RELATED INFORMATION

`atm_cmm_activate_con`, `atm_cmm_con_release`, `xxx_add`, `xxx_setup`

Section 7.4.2 for information on releasing connections

`atm_cmm_con_failed`

NAME

`atm_cmm_con_failed` – Notifies the CMM of a call failure

SYNOPSIS

```
atm_error_t atm_cmm_con_failed(  
    atm_sig_handle_t sm,  
    atm_addr_t *addr);
```

ARGUMENTS

- sm* Specifies the signaling module handle that the CMM returned in the registration call.
- addr* Specifies a pointer to an `atm_addr` structure that contains all the parameters of the call. This pointer must point to the same `atm_addr` structure that was passed in either the `atm_cmm_new_call` function call or in the `xxx_setup` and `xxx_add` function calls since this pointer is the handle that refers to the specific connection.

DESCRIPTION

The `atm_cmm_con_failed` routine is a signaling module interface. When the CMM makes a connection request to a signaling module, the call to the remote system might fail for some reason. Since call creation is an asynchronous operation, the signaling module might encounter errors during call processing after the `xxx_setup` call has returned. In these cases, the signaling module notifies the CMM of failures both of point-to-point connections and of additional endpoints in point-to-multipoint connections. In the case of the failure of endpoints in a point-to-multipoint connection, the `addr` pointer points to the `atm_addr` structure of the added endpoint.

RETURN VALUES

If the endpoint referenced is invalid, the `atm_cmm_con_failed` routine returns `ATM_CAUSE_EIR` to indicate that it has no previous entry for the connection; otherwise, it returns `ATM_CAUSE_GOOD`.

`atm_cmm_con_failed`

RELATED INFORMATION

Section 7.1 for information on setting up connections

`atm_cmm_con_release`

NAME

`atm_cmm_con_release` – Notifies the CMM that a connection will be released

SYNOPSIS

```
atm_error_t atm_cmm_con_release(  
    atm_sig_handle_t sm,  
    unsigned long reference,  
    atm_ppa_p ppa);
```

ARGUMENTS

<i>sm</i>	Specifies the signaling module handle that the CMM returned in the registration call.
<i>reference</i>	Specifies the unique call reference value that identifies the connection to be released.
<i>ppa</i>	Specifies a pointer to the PPA to which the VC being released belongs.

DESCRIPTION

The `atm_cmm_con_release` routine is a signaling module interface. The signaling module calls this routine when the module receives a request to tear down a connection from the network or endpoint. This notifies the CMM to release a connection or virtual circuit, and not to drop a single endpoint from a connection. When this routine is called, the CMM makes the connection unavailable for transmission, initiates the teardown of the referenced connection and all associated endpoints, and awaits notification that the VC has been torn down.

RETURN VALUES

If the endpoint referenced is invalid, the `atm_cmm_con_release` routine returns `ATM_CAUSE_EIR` to indicate that it has no previous entry for the connection; otherwise, it returns `ATM_CAUSE_GOOD`.

`atm_cmm_con_release`

RELATED INFORMATION

`atm_cmm_activate_con`, `atm_cmm_con_deleted`

Section 7.4 for information on releasing connections

`atm_cmm_connect`

NAME

`atm_cmm_connect` – Requests a connection to a remote system

SYNOPSIS

```
atm_error_t atm_cmm_connect(  
    atm_cvg_handle_t cm,  
    enum atm_ctype type,  
    atm_bind_handle_t calling,  
    union atm_cmi_addr called,  
    atm_uni_call_ie_p ei,  
    int aging,  
    atm_vc_service_p params);
```

ARGUMENTS

cm Specifies a value returned to the convergence module by the registration function call. This uniquely identifies the convergence module making the request.

type Specifies the type of connection being requested. The following types are defined:

Connection	Meaning
ATM_CT_PTM	Specifies that the connection is the first connection in a point-to-multipoint connection.
ATM_CT_PTP	Specifies that the connection is a point-to-point connection only.
ATM_CT_PVC	Specifies that the connection is a permanent virtual circuit.

calling Specifies the calling party address information to be used when placing the call. This is the identifier returned from the bind operation when the convergence module is bound to a PPA. This specifies the fully qualified AESA that will be used as the calling party address, and is used to direct the call to the proper signaling module and device driver.

`atm_cmm_connect`

- called* Specifies the called party (destination) address for the connection. When requesting the creation of a new SVC, the `atm_cmi_addr` union contains a pointer to an `atm_addr` structure that the convergence module allocates (using the proper CMM routine) with all necessary structure members set to specify the intended destination of the call. The convergence module must not free the storage for this structure; the CMM performs the deallocation when the connection closes (the caller is responsible for deallocating this structure if this routine returns an error indication). When requesting the creation of a new PVC, this argument contains the VCI and VPI for the new VC. Interpretation of the information in this argument is based on the value of the *type* argument. See Section 6.22.2 for information on the `atm_cmi_addr` union.
- ie* Specifies a pointer to an array of `atm_uni_call_ie` structures that has been initialized with information to be used in placing the call. This argument must be NULL if no optional IEs are specified.
- aging* Specifies how the CMM is to age the connection. The CMM provides connection aging services to detect connections that are unused and to delete the resources associated with the connections. The CMM provides several aging algorithms. The convergence modules must specify an initial aging algorithm at the time the circuit is created, but can modify this setting at any time through the `atm_cmm_vc_control` routine.
- params* Specifies an `atm_vc_services_p` pointing to a properly set `atm_vc_services` structure that defines the circuit parameters. For CBR or pacing circuits, the *params* argument can be a resource reservation, that is, a services structure that is backed by reserved resources.

DESCRIPTION

The `atm_cmm_connect` routine is a convergence module interface that requests the CMM to create a connection to a remote system. This initiates the exchanges between the network and the remote host to create a new

`atm_cmm_connect`

connection. The convergence module must specify all connection parameters to the CMM; these parameters are required to create the new connection.

By requesting a connection to a remote host, the convergence module owns the VC; the VC's ownership cannot be changed. The convergence module is notified of all incoming data and exceptions on the VC. Also, the convergence module is the only module that can transmit data on the VC.

NOTES

To create a PVC, set the *type* argument to `ATM_CT_PVC`, set the *calling* argument to the bind handle returned by the CMM for a binding on the PVC PPA, set the *params* argument to the circuit parameters for the new PVC, and set the *called* argument to an `atm_cmi_addr` union that contains valid *vci* and *vpi* data; the *aging* arguments are ignored.

If the system cannot provide the level of service specified in the *params* argument, the appropriate error is returned. No parameter negotiation is done for PVCs.

EXAMPLE

See Section B.1 for a code example of the `atm_cmm_connect` routine.

RETURN VALUES

If the call is proceeding, the `atm_cmm_connect` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number that indicates the reason the call cannot proceed.

If the call proceeds, the `atm_vc_p` information in the `atm_addr` structure is valid on return. However, this does not mean the VC is ready to carry data. It means only that the CMM has allocated an `atm_vc` structure with which it can keep track of the connection. The convergence module is notified through the `xxx_except` routine when the call has completed and when the VC can carry data.

RELATED INFORMATION

`atm_cmm_vc_control`

Section 6.22.2 for information on the `atm_cmi_addr` union

atm_cmm_connect

Section 7.3 for information on connection aging

Section 7.5 for information on PVC creation

`atm_cmm_del_esi`

NAME

`atm_cmm_del_esi` – Deletes an ATM address

SYNOPSIS

```
atm_err_t atm_cmm_del_esi(  
    atm_cvg_handle_t,  
    atm_esi_handle_t);
```

DESCRIPTION

The `atm_cmm_del_esi` routine is a convergence module interface that deletes an ATM address. A convergence module uses this routine when it no longer needs a new address it has created. Only the convergence module that registered the address can unregister the address.

Deleting an ESI causes the signaling module to delete all PPAs created from the ESI.

Deleting a PPA causes all VCs associated with that PPA to be closed by the CMM (both ends of the connection receive proper notification). Thus, convergence modules should delete addresses only when there are no active VCs associated with the address's PPAs.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_new_esi`

`atm_cmm_del_ppa`

NAME

`atm_cmm_del_ppa` – Notifies the CMM that a deleted PPA is invalid

SYNOPSIS

```
atm_error_t atm_cmm_del_ppa(  
    atm_sig_handle_t sm,  
    atm_ppa_p ppa);
```

ARGUMENTS

- | | |
|------------|---|
| <i>sm</i> | Specifies the signaling module handle that the CMM returned in the registration call. |
| <i>ppa</i> | Specifies the PPA pointer that was returned when the PPA was added to the system. |

DESCRIPTION

The `atm_cmm_del_ppa` routine is a signaling module interface. When a signaling protocol, in cooperation with a switch, deletes an address from the list of recognized addresses on an interface, the signaling module calls this routine to notify the CMM that the deleted PPA associated with the address is no longer valid. The CMM then informs convergence modules bound to the PPA that the address is no longer valid and initiates a teardown of all VCs associated with the address.

All PPAs, except for the PVC PPAs, are owned by a signaling module. That is, a signaling module is always responsible for the creation and deletion of a PPA. This is required since the registration of addresses with a switch is handled entirely by signaling protocols. Also, PPAs may be deleted because of actions on the network that are completely unrelated to the local system. Because of this, the CMM does not automatically delete PPAs when an interface is taken down or loses its connection to the switch. The CMM responds to an interface shutdown by deleting the PVC PPA. Signaling modules will delete PPAs when it is appropriate for their protocols to do so (such as when they lose communications with the switch).

`atm_cmm_del_ppa`

RETURN VALUES

Upon successful completion, the `atm_cmm_del_ppa` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number.

RELATED INFORMATION

`atm_cmm_new_ppa`, `atm_cmm_ppa_bind`, `atm_cmm_ppa_info`

`atm_cmm_drop`

NAME

`atm_cmm_drop` – Drops a connection to an endpoint

SYNOPSIS

```
atm_err_t atm_cmm_drop(  
    atm_cvg_handle_t cm,  
    atm_addr_t *addr);
```

ARGUMENTS

- | | |
|-------------|--|
| <i>cm</i> | Specifies a value that the registration function call returns to the convergence module. This uniquely identifies the convergence module making the request. |
| <i>addr</i> | Specifies a pointer to the <code>atm_addr</code> structure of the endpoint to be dropped. |

DESCRIPTION

The `atm_cmm_drop` routine is a convergence module interface that drops a connection to an endpoint. The endpoint can be associated with either a point-to-point connection or a point-to-multipoint connection. When the last endpoint associated with a VC is dropped, the VC is torn down. You use this routine to manage multipoint VCs; you can also use it to initiate the teardown of point-to-point VCs.

RETURN VALUES

If the endpoint teardown is proceeding, the `atm_cmm_drop` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number. After the endpoint is torn down, the convergence module receives an `ATM_CME_EP_DEAD` exception notification. When the last endpoint associated with a VC has been disconnected, the VC is torn down.

RELATED INFORMATION

`atm_cmm_add`

`atm_cmm_enquery`

NAME

`atm_cmm_enquery` – Requests that connection state information for an endpoint be updated

SYNOPSIS

```
atm_err_t atm_cmm_enquery(  
    atm_cvg_handle_t cm,  
    atm_addr_t *ep);
```

ARGUMENTS

- cm* Specifies a value that the registration function call returns to the convergence module. This uniquely identifies the convergence module making the request.
- ep* Specifies the endpoint to be queried.

DESCRIPTION

The `atm_cmm_enquery` routine is a convergence module interface that requests that the connection state information for an endpoint be updated. This initiates a connection enquiry to the endpoint that results in the `atm_addr` structure for the endpoint being updated with the latest status information. This routine may be called any time a connection is active.

RETURN VALUES

If the enquiry is initiated, the `atm_cmm_enquery` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number, indicating the reason the enquiry was not initiated. When the enquiry reply is received by the system, the convergence module is informed by an `ATM_CME_ENQUERY_DONE` exception notification. When the convergence module gets this notification, it may then examine the updated endpoint status information in the `atm_addr` structure.

RELATED INFORMATION

`atm_cmm_add`

atm_cmm_ep_dropped

NAME

atm_cmm_ep_dropped – Notifies the CMM to drop an endpoint

SYNOPSIS

```
atm_err_t atm_cmm_ep_dropped(  
    atm_sig_handle_t sm,  
    unsigned long reference,  
    long epreference,  
    atm_ppa_p ppa);
```

ARGUMENTS

<i>sm</i>	Specifies the signaling module handle that the CMM returned in the registration call.
<i>reference</i>	Specifies the unique call reference value that identifies the connection to be released.
<i>epreference</i>	Specifies the unique endpoint reference value that identifies the endpoint to be dropped.
<i>ppa</i>	Specifies a pointer to the PPA to which the VC being released belongs.

DESCRIPTION

The `atm_cmm_ep_dropped` routine is a signaling module interface. A signaling module calls this routine when the module receives a request to drop an endpoint from a connection. This notifies the CMM to drop endpoints in a multipoint connection. The endpoint must already exist on the system.

The CMM notifies the convergence module that owns the endpoint's VC that the endpoint has been dropped and deletes the endpoint. If the last endpoint associated with a VC is dropped, the CMM initiates the release of the VC.

`atm_cmm_ep_dropped`

RETURN VALUES

If the endpoint referenced is invalid, the `atm_cmm_ep_dropped` routine returns `ATM_CAUSE_EIR` to indicate that it has no previous entry for the connection; otherwise, it returns `ATM_CAUSE_GOOD`.

RELATED INFORMATION

`atm_cmm_add`, `atm_cmm_drop`

`atm_cmm_error`

NAME

`atm_cmm_error` – Reports errors to the CMM

SYNOPSIS

```
void atm_cmm_error(  
    atm_drv_handle_t driver,  
    int error,  
    atm_vc_t *vc);
```

ARGUMENTS

driver Specifies the interface on which the error has occurred. This is the handle assigned to the driver when the driver registers with the CMM.

error Specifies a numeric value that indicates one of the following error types:

Error	Meaning
ATM_DE_DOWN	The driver detected a fatal medium error. In response, the CMM initiates the destruction of all VCs belonging to the failed interface. Further, the CMM considers the interface unavailable and rejects any requests for new connections on the interface. The driver must inform the CMM when the media is functional again. The <i>vc</i> argument is ignored for this notification.
ATM_DE_UP	The driver detected that the medium is up and ready to transfer data. When the CMM receives this indication, it allows the processing of connection requests. The <i>vc</i> argument is ignored for this notification.

atm_cmm_error

Error	Meaning
ATM_DE_STARTQ	A driver output queue whose flow was controlled is now able to accept additional data for transmission. The <i>vc</i> argument is the number of the queue that is being enabled ($0 \leq vc < atm_drv_params.nqueue$).
ATM_DE_VC_FATAL	A local, fatal error has occurred to the interface that causes one or more VCs to become inoperative. An example of this type of error might be an adapter memory error that destroys any information about the VC on the adapter. This indication reports errors from which the driver is unable to recover. The <i>vc</i> argument is a pointer to the failed VC.

vc Specifies a pointer to the VC on which the error occurred. If the error is not associated with a VC, this is NULL.

DESCRIPTION

The `atm_cmm_error` routine is a device driver interface. When an ATM device driver detects errors on VCs or other interface failures, the module calls this routine to report the error to the CMM.

When the CMM receives the error report, it recovers or shuts down the VC with the errors. The driver does not need to perform any other actions. If the VC must be destroyed, the CMM calls the driver through the driver management interface to deactivate or destroy the VC. If the error indicates an interface failure, the CMM tears down VCs on the interface.

RETURN VALUES

None

RELATED INFORMATION

Chapter 10 for information on flow control in the ATM subsystem

`atm_cmm_findaddr`

NAME

`atm_cmm_findaddr` – Requests endpoint and VC information from the CMM

SYNOPSIS

```
atm_addr_p atm_cmm_findaddr(  
    long call_reference,  
    long endpoint,  
    atm_ppa_p ppa);
```

ARGUMENTS

<i>call_reference</i>	Specifies the unique call reference value assigned to the VC.
<i>endpoint</i>	Specifies the endpoint reference value for an endpoint in a point-to-multipoint connection. This argument is set to 0 for the root of point-to-multipoint connections and for point-to-point connections.
<i>ppa</i>	Specifies the interface associated with the endpoint to look up. Since call reference values are usually unique only per interface, you must specify this argument to differentiate between VCs with the same reference value but belonging to different interfaces.

DESCRIPTION

The `atm_cmm_findaddr` routine is a signaling module interface that signaling modules use to request endpoint and VC information from the CMM. The CMM maintains all state information about each VC and calls (endpoints) associated with the VC.

Signaling modules can call this routine at any time to resolve a reference to a connection endpoint. Once the reference is resolved, the signaling module can access and modify structures as necessary as long as you follow locking conventions. For example, if the signaling module needs to process a

`atm_cmm_findaddr`

STATUS ENQUIRY request, all the information necessary to format the STATUS reply is in the `atm_addr` structure. This routine returns a reference to the structure, enabling the signaling module to extract the data needed to format the reply.

RETURN VALUES

Upon successful completion, the `atm_cmm_findaddr` call returns a pointer to the `atm_addr` structure for the specified endpoint. Signaling modules can use this information to find the `atm_vc` structure, if necessary. If the specified endpoint is not found, the CMM returns NULL.

RELATED INFORMATION

`atm_cmm_add`

`atm_cmm_find_driver`

NAME

`atm_cmm_find_driver` – Determines whether a driver is registered with the CMM

SYNOPSIS

```
atm_drv_handle_t atm_cmm_find_driver(  
    char *name);
```

ARGUMENTS

name Specifies a NULL-terminated character string that identifies the driver.

DESCRIPTION

The `atm_cmm_find_driver` routine is a convergence and signaling module interface that enables the module to determine whether a specified driver is registered with the CMM.

If a convergence or signaling module provides MMI services, the module can use this routine to determine whether the driver over which it is requested to operate actually exists.

RETURN VALUES

Upon successful completion, the `atm_cmm_find_driver` call returns the handle (`atm_drv_handle_p`) that identifies the driver to the CMM. If the driver is not registered, the call returns NULL.

RELATED INFORMATION

`atm_cmm_register_dd`

`atm_cmm_free_addr`

NAME

`atm_cmm_free_addr` – Frees memory allocated to `atm_addr` structures

SYNOPSIS

```
void atm_cmm_free_addr(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure to free. This is the same pointer passed into the `atm_alloc_addr` routine.

DESCRIPTION

The `atm_cmm_free_addr` routine is a convergence module interface that convergence modules use to free memory that they have allocated.

Usually, the CMM frees all ATM address structures associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails), it may be necessary for the allocating module to free memory it has allocated. In these cases, modules can call the `atm_cmm_free_addr` routine with the value returned from `atm_cmm_alloc_addr` to free memory.

RETURN VALUES

None.

RELATED INFORMATION

`atm_cmm_alloc_addr`

`atm_cmm_free_ie`

NAME

`atm_cmm_free_ie` – Frees memory allocated to IE structures

SYNOPSIS

```
void atm_cmm_free_ie(  
    atm_uni_call_ie_p u);
```

ARGUMENTS

`u` Specifies a pointer to the `atm_call_ie` structure to free. This is the same pointer returned from the `atm_cmm_alloc_ie` routine as a result of a successful storage allocation.

DESCRIPTION

The `atm_cmm_free_ie` routine is a convergence module interface that convergence modules use to free IE structure memory that they have allocated.

EXAMPLE

See Section B.1 for a code example of the `atm_cmm_free_ie` routine.

RETURN VALUES

None.

RELATED INFORMATION

`atm_cmm_alloc_ie`

`atm_cmm_free_services`

NAME

`atm_cmm_free_services` – Frees memory allocated to `atm_vc_services` structures and releases reserved resources

SYNOPSIS

```
atm_error_t atm_cmm_free_services (  
    atm_vc_services_p s);
```

ARGUMENTS

s Specifies a pointer to the `atm_vc_services` structure to free. This is the same pointer passed into the `atm_alloc_services` routine.

DESCRIPTION

The `atm_cmm_free_services` routine is a convergence module interface that convergence modules use to free memory that they have allocated. Typically, the CMM frees the `atm_vc_services` structure associated with a VC when the VC is destroyed. However, under some error conditions (such as when a convergence module is allocating a series of structures and one allocation fails), the allocating module might need to free memory it has allocated. In these cases, modules call the `atm_cmm_free_services` routine with the value returned from `atm_cmm_alloc_services` to free memory.

A convergence module can also use the `atm_cmm_free_services` call to free a resource reservation or to terminate a resource reservation request. (See Section 7.1 for more information.) The `atm_cmm_free_services` call frees the `atm_vc_services_t` memory and releases its reserved resources, if any.

RETURN VALUES

If the CMM is able to free the memory and any reserved resources, the `atm_cmm_free_services` call returns `ATM_CAUSE_GOOD`. If the services structure is associated with a resource that is in the process of being released, the call returns `ATM_CAUSE_BUSY`. If the convergence module

atm_cmm_free_services

inadvertently attempts to free an `atm_vc_services_t` that is attached to an active connection, `ATM_CAUSE_BUSY` is also returned.

RELATED INFORMATION

`atm_cmm_alloc_services`, `atm_cmm_reserve_resources`, `xxx_except`

Section 6.2.1 for information on convergence module exception handling

atm_cmm_mib_response

NAME

`atm_cmm_mib_response` – Notifies the CMM of the completion of a MIB access

SYNOPSIS

```
atm_error_t atm_cmm_mib_response(  
    void *id,  
    unsigned int fieldlen,  
    unsigned char *fieldval);
```

ARGUMENTS

<i>id</i>	Specifies the access ID that the CMM passes into the signaling module's MIB access routine. The signaling module must not modify this value.
<i>fieldlen</i>	Specifies the length (in bytes) of the returned information. This argument is valid only in response to an <code>ATM_MIB_GET</code> operation.
<i>fieldval</i>	Specifies a pointer to a series of bytes that constitute the value being retrieved by an <code>ATM_MIB_GET</code> operation. Do not use this for an <code>ATM_MIB_GET</code> operation.

DESCRIPTION

The `atm_cmm_mib_response` routine is a signaling module interface. Sometimes a signaling module's MIB access routine can block because of the need to contact some other network management entity. When the signaling module returns from the MIB access call, the module calls this routine to notify the CMM of the completion of a MIB access and report any retrieved values.

RETURN VALUES

Upon successful completion, the `atm_cmm_mib_response` call returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number.

`atm_cmm_mib_response`

RELATED INFORMATION

`atm_cmm_register_sig, xxx_mib`

`atm_cmm_new_call`

NAME

`atm_cmm_new_call` – Notifies CMM of a connection request

SYNOPSIS

```
atm_error_t atm_cmm_new_call(  
    atm_sig_handle_t sm,  
    atm_addr_t *addr,  
    atm_uni_call_ie_p ie,  
    atm_vc_services services,  
    atm_ppa_p ppap,  
    int selector,  
    unsigned int vpi,  
    unsigned int vci,  
    long call_reference);
```

ARGUMENTS

<i>sm</i>	Specifies the signaling protocol module handle that was returned by the registration function.
<i>addr</i>	Specifies a pointer to an <code>atm_addr</code> structure that contains all the information about the incoming call. The signaling module writes the information into the structure.
<i>ie</i>	Specifies a pointer to an array of optional information elements received from the calling party. The signaling module converts data from the call setup message and fills in an array of information element structures.
<i>services</i>	Specifies a pointer to a properly set <code>atm_vc_services</code> structure. This indicates the type of service the calling party is requesting. The signaling protocol module must extract the required information from its signaling protocol messages. All members of this structure must be set. If a signaling protocol does not provide means by which some of the needed values can be specified, the signaling protocol module must supply some

`atm_cmm_new_call`

reasonable defaults. The CMM and driver require this information to determine if there are sufficient local resources to handle the services the caller is requesting. The CMM modifies values in this structure to indicate that the value should be negotiated with the calling party. (This usually means that the signaling module simply returns the modified values in the reply to the calling party.)

<i>ppap</i>	Specifies the PPA on which the incoming call arrived.
<i>selector</i>	Specifies the selector value taken from the called party address information sent by the caller. This indicates the service on the PPA that is being called.
<i>vpi</i>	Specifies the VPI value for the new VC that the network has created for this call.
<i>vci</i>	Specifies the VCI value for the new VC that the network has created for this call.
<i>call_reference</i>	Specifies a unique value that the signaling module has assigned to this call. This is the value used to reference this call during the call's entire lifetime on the system. This value is opaque to the CMM and can have special meaning to the signaling protocol module.

DESCRIPTION

The `atm_cmm_new_call` routine is a signaling module interface that a signaling module calls when it receives a new call. This notifies the CMM of the new call. The CMM determines whether the call should be accepted. This CMM routine is nonblocking and returns with an indication about the disposition of the call.

`atm_cmm_new_call`

RETURN VALUES

The CMM evaluates the call parameters and determines if the call should be accepted. If the call is accepted, the `atm_cmm_new_call` routine returns `ATM_CAUSE_GOOD`. If the CMM and convergence module cannot immediately decide on the disposition of the call, `ATM_CAUSE_DEFER` is returned; the signaling module's exception routine is called when the disposition of the call is determined.

If the call is rejected, this routine returns an ATM error number indicating the reason. The convergence module may supply additional information in the `cause` fields in the `atm_addr` structure. If the call is rejected (a value other than `ATM_CAUSE_GOOD` or `ATM_CAUSE_DEFER` is returned), the signaling module must deallocate the storage for the `atm_addr` structure, the information elements, and the `atm_vc_services` structure.

RELATED INFORMATION

`xxx_except`

Section 7.2 for information on receiving connections

`atm_cmm_new_esi`

NAME

`atm_cmm_new_esi` – Registers an ATM address with the network

SYNOPSIS

```
atm_esi_handle_t atm_cmm_new_esi(
    atm_cvg_handle_t cvg_handle,
    unsigned char *esi,
    int esilen,
    atm_ppa_p driver,
    void *arg);
```

ARGUMENTS

<i>cvg_handle</i>	Specifies the convergence module handle returned to the convergence module when it registered with the CMM.
<i>esi</i>	Specifies a pointer to an array of the 6 bytes of ESI to be registered with the network.
<i>esilen</i>	Specifies the size (in bytes) of the ESI being added. This must be less than <code>ATM_MAX_ESI</code> . For UNI signaling, this is always 6.
<i>driver</i>	Specifies the driver handle of the interface on which the address is to be registered. This value is obtained from the PVC PPA for the driver on which the ESI is being registered.
<i>arg</i>	Specifies a value that the convergence module creating the ESI uses to identify any PPAs created from the ESI. The CMM does not modify the value of this argument.

DESCRIPTION

The `atm_cmm_new_esi` routine is a convergence module interface that registers an ATM address with the network. The module supplies the ESI (or other signaling protocol equivalent) portion of the address and a driver

`atm_cmm_new_esi`

reference to the CMM. The CMM then performs all the necessary functions needed to register the address with the network and to activate it.

When a convergence module supplies a new ESI, the signaling modules apply the ESI in a signaling protocol-specific way. Depending on the signaling protocol, the addition of a single ESI to the system could create many PPAs.

The calling convergence module is notified of the new PPA(s) when the registration is complete. Note that the addition of a new ESI does not cause the immediate creation of a new PPA in cases where there is no signaling protocol that accepts the ESI, or if the link to the switch is down. In these cases, new PPAs are created when new signaling modules are registered and accept the ESIs, or when links to the switch become functional.

Once a convergence module creates an address, only the creating convergence module, the CMM (through the system `atmconfig` program), or the network can delete the addresses; other convergence modules cannot delete the address. The creating convergence module must issue the `atm_cmm_ppa_bind` call once it receives notification that the PPA(s) have been configured. If the convergence module unregisters, all its created addresses are destroyed.

If an ESI is deleted, the calling convergence module receives an `ATM_CME_ESI_DEL` exception notification.

RETURN VALUES

If the registration process is proceeding, the `atm_cmm_new_esi` routine returns a handle for the new ESI; otherwise, it returns a NULL. If the requested ESI is already registered, the routine also returns NULL.

RELATED INFORMATION

`atm_cmm_del_esi`, `atm_cmm_ppa_bind`

`atm_cmm_new_ppa`

NAME

`atm_cmm_new_ppa` – Notifies the CMM that a new PPA exists

SYNOPSIS

```
atm_ppa_p atm_cmm_new_ppa(  
    atm_sig_handle_t sm,  
    unsigned char *addr,  
    unsigned int addrlen,  
    unsigned char ton,  
    unsigned char anpi,  
    atm_drv_handle_t driver,  
    atm_esi_p esi,  
    void *ppas_id,  
    atm_uni_type_t uni,  
    void *sig_info);
```

ARGUMENTS

- | | |
|----------------|---|
| <i>sm</i> | Specifies the signaling module handle that the registration function call returned. |
| <i>addr</i> | Specifies a pointer to a series of bytes in memory that is the fully formed new address. The CMM copies this information and the signaling module can free these bytes on return if they are not needed for internal use. |
| <i>addrlen</i> | Specifies the exact number of bytes in the address. This must be less than <code>ATM_PPA_MAX_ADDR</code> . For UNI signaling, the value of this argument is always 20. |
| <i>ton</i> | Specifies the endpoint's address type. These bits are the same as the type of number field in the Called Party Number IE (right justified). |
| <i>anpi</i> | Specifies the address or numbering plan identification information for the address. These bits are the same as those in the corresponding field of the Called Party Number IE. |
| <i>driver</i> | Specifies the device driver handle for the interface on which the new address has been created. This value is taken from |

`atm_cmm_new_ppa`

the PVC PPA over which the signaling module has created the new address.

- esi* Specifies which ESI was used in the creation of this address. This is the `atm_esi` structure pointer that gets passed in to the signaling module on the `ATM_SIGE_NEWESI` exception.
- ppas_id* Specifies an internal value that the signaling module can use to keep track of the PPA. The CMM does not use or modify the value of this argument, which gets placed in the `atm_ppa` structure for use by the signaling module.
- uni* Identifies the type of UNI (public or private) associated with the PPA.
- sig_info* Identifies a piece of data that the signaling module needs to associate with the PPA. For UNI signaling, set this value to the UNI version that is in use on the PPA. The version is specified as an `atm_uni_vers_t` constant.

DESCRIPTION

The `atm_cmm_new_ppa` routine is a signaling module interface. When a signaling module creates a new address (new PPA), the module calls this routine. This notifies the CMM that the new PPA exists. A convergence module can use the PPA to make and receive calls.

For example, when a UNI 3.0/3.1 signaling module is informed of a new address prefix, through the Interim Local Management Interface (ILMI), that has been created on the switch, the module combines the new prefix with existing end system identifiers (ESIs) to form a new set of addresses for the new prefix. Then, the signaling module tells the CMM about each of these new addresses.

RETURN VALUES

Upon successful completion, the `atm_cmm_new_ppa` routine returns a valid PPA pointer for the new PPA; otherwise, it returns NULL.

`atm_cmm_new_ppa`

RELATED INFORMATION

`atm_cmm_del_ppa`, `atm_cmm_ppa_bind`, `atm_cmm_ppa_info`

NAME

atm_cmm_new_thread – Creates a kernel thread for use by an ATM module

SYNOPSIS

```
thread_t atm_cmm_new_thread(  
    task_t task,  
    void(*) (void * arg) *first,  
    void *arg,  
    int pri,  
    int maxpri);
```

ARGUMENTS

- | | |
|---------------|---|
| <i>task</i> | Specifies a pointer to a task structure used to uniquely identify the thread. The contents of this structure are filled in by this routine and should never be modified. This argument must reference a structure that will be valid for the life of the thread (generally a global structure). |
| <i>first</i> | Specifies the address of the routine that serves as the thread's entry point. This is the first routine called in the new thread's context. Returning from this routine terminates the thread. This routine is passed a single argument whose use is entirely thread specific. |
| <i>arg</i> | Specifies the argument that is passed as the only argument to the thread's entry point. The use and interpretation of this argument is entirely up to the module creating and using the thread. |
| <i>pri</i> | Specifies the priority at which the thread runs when its entry point is called. This is also the priority at which the thread always runs unless you change the priority by using one of the kernel thread management functions. |
| <i>maxpri</i> | Specifies the maximum priority at which the thread is allowed to run. You cannot set the thread priority higher than this value. Note that thread priorities range from 1 (the highest) to |

atm_cmm_new_thread

63 (the lowest). Set the thread priority as follows: *maxpri* <= *newpri* <= 63.

DESCRIPTION

The `atm_cmm_new_thread` routine is a signaling, convergence, and device driver module interface. When an ATM module creates a new thread for use by any ATM module, the module calls this routine. This routine combines many of the DIGITAL UNIX thread primitives to provide an easier interface for ATM modules.

RETURN VALUES

Upon successful completion, the `atm_cmm_new_thread` routine returns a pointer to a thread structure that is used to reference the thread in all subsequent calls to kernel thread routines. If a thread could not be created, it returns NULL.

RELATED INFORMATION

See Section 3.8.4 for a description of ATM thread creation.

`atm_cmm_next_cause`

NAME

`atm_cmm_next_cause` – Retrieves logged VC or endpoint cause information

SYNOPSIS

```
atm_cause_info_p atm_cmm_next_cause(  
    atm_vc_p vp,  
    atm_addr_p addr,  
    atm_cause_info_p cip);
```

ARGUMENTS

- vp* Specifies a pointer to the VC from which to retrieve cause information. If you want information for only one endpoint, set the value to NULL and set the *addr* value to non-NULL.
- addr* Specifies a pointer to the `atm_addr` structure for the endpoint from which to retrieve cause information. If the error applies to all endpoints on the VC (global), set the value to NULL and set the *vp* value to non-NULL.
- cip* Specifies the `atm_cause_info_p` value returned by the previous call to `atm_cmm_next_cause`. To retrieve the first cause entry for a VC or endpoint, set the value to NULL.

DESCRIPTION

The `atm_cmm_next_cause` routine is a convergence and signaling module interface that retrieves error or normal condition information logged by previous calls to `atm_cmm_set_cause`, `atm_cmm_adi_set_cause`, or `atm_cmm_smi_set_cause`.

To retrieve the first logged cause for a VC or endpoint, set the value of the *cip* parameter to NULL. To retrieve subsequent causes, set the value of the *cip* parameter to the value returned from the previous call to `atm_cmm_next_cause`.

`atm_cmm_next_cause`

RETURN VALUES

Upon successful completion, the `atm_cmm_next_cause` routine returns a pointer to the next `atm_cmm_next_cause` structure for the specified VC or endpoint. If both the VC (*vp*) and endpoint (*addr*) arguments are NULL, or if there are no more causes for the specified VC or endpoint, the routine returns NULL.

RELATED INFORMATION

`atm_cmm_adi_set_cause`, `atm_cmm_set_cause`,
`atm_cmm_smi_set_cause`

Section 3.10.7 for information on cause information

`atm_cmm_oam_receive`

NAME

`atm_cmm_oam_receive` – Passes OAM cells to the CMM

SYNOPSIS

```
void atm_cmm_oam_receive(  
    atm_drv_handle_t driver,  
    struct mbuf *mbp,  
    atm_vc_t *vc);
```

ARGUMENTS

- driver* Specifies the interface on which the operations and maintenance (OAM) cell was received. This is the driver handle that the CMM assigned at driver registration time.
- mbp* Specifies a pointer to a single mbuf that contains the complete OAM cell (including the cell header). You can pass only one OAM cell to the CMM in a single call.
- vc* Specifies a pointer to the VC to which the OAM cell belongs. The CMM creates PVCs for VCs 3 and 4 for the receipt of F4 OAM flows.

DESCRIPTION

The `atm_cmm_oam_receive` routine is a device driver interface. When an ATM device driver receives data packets and cells from the ATM network, the driver calls this routine to pass the data to the CMM. Device drivers should pass all OAM cells to the CMM.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_receive`

Section 3.5.3 for information on how data are carried

atm_cmm_ppa_bind

NAME

atm_cmm_ppa_bind – Binds a convergence module to a PPA

SYNOPSIS

```
atm_bind_handle_t atm_cmm_ppa_bind(  
    atm_ppa_p ppa,  
    atm_cvg_handle_t cvg_handle,  
    int selector,  
    void *const bind_handle,  
    atm_error_t (*const xxx_connect)(void bind_handle,  
        atm_addr_p addr,  
        atm_bind_handle_t myaddr,  
        atm_vc_p vc,  
        atm_uni_call_ie_p *reply,  
        atm_vc_services_p requested,  
        atm_vc_services_p *avail));
```

ARGUMENTS

<i>ppa</i>	Specifies a pointer to the <code>atm_ppa</code> structure of the PPA to which the convergence module is binding. This is the same pointer passed into the convergence module when it is notified of the existence of the new PPA.
<i>cvg_handle</i>	Specifies the convergence module handle returned to the convergence module when it registered with the CMM. This argument must be valid for the bind to proceed.
<i>selector</i>	Specifies the selector byte that the convergence module will use as its bind point. This value must not already be bound to an AESA. If this value is <code>ATM_CMM_BIND_PICK</code> , the CMM assigns the selector value. If the caller specifies the selector value to use, it must be in the range of 0 to 255 (for binding to SVC PPAs) and the high-order bit must be clear (for binding to PVC PPAs).
<i>bind_handle</i>	Specifies a value that the CMM will pass back to the convergence module when it informs the

`atm_cmm_ppa_bind`

convergence module of a new call. The convergence module can use this value to identify any internal references to the bound PPA. This value has no meaning to the CMM and is not modified by the CMM.

`xxx_connect`

Specifies a pointer to a routine that the CMM is to call when a new connection is being processed. If a convergence module should not receive a connection through a bind point, the value of this argument must be NULL.

DESCRIPTION

The `atm_cmm_ppa_bind` routine is a convergence module interface that binds a convergence module to a PPA. A convergence module must call this routine if it wants to make outgoing calls or to receive incoming calls being made to one of the configured local ATM addresses. The routine informs the CMM that the convergence module will function as a network service user for a specific address and selector value.

Convergence modules are notified when a PPA is configured on the system (when an address is registered with a switch and is made known to the ATM network). Convergence modules can then bind to the PPA to create a network service user endpoint, which uniquely identifies the convergence module on the network. Once bound, there is a unique AESA associated with the bound convergence module. The AESA and the bind point identify the service on the network and local system. Convergence modules will be informed only of incoming calls on PPAs to which they have bound and can place calls only through a bind point. The CMM handles calls only to AESAs. If the AESA specified by the called party address of an incoming call does not exist, the CMM rejects the call.

By binding to a PPA, a convergence module is creating an AESA and uniquely identifying its service on the network. Once this is done, incoming calls can be routed to the convergence module, and the convergence module can make outgoing calls. All call activity is directed at a bind point, and thus, a specific convergence module.

When binding to a PPA, a convergence module can either specify a specific selector value it wants to use in creating the AESA address, or it may

`atm_cmm_ppa_bind`

allow the CMM to assign it a selector value. In both cases, the selector value must specify a unique endpoint on the PPA.

Note that a convergence module can bind to up to 256 times to a PPA (as long as all its selector values are unique), and may bind to as many PPAs as is necessary to provide its service.

RETURN VALUES

Upon successful completion, the `atm_cmm_ppa_bind` routine returns a value that the convergence module uses to reference the bind point; otherwise, it returns a NULL. Typically, bind failure occurs for the following reasons: insufficient memory, the PPA is no longer valid, the selector value specified is in use, or there are no free selectors available.

RELATED INFORMATION

`atm_cmm_bind_info`, `atm_cmm_del_ppa`, `atm_cmm_new_ppa`,
`atm_cmm_ppa_info`, `atm_cmm_ppa_unbind`, `xxx_connect`

Section 3.6 for information on PPAs and AESAs

`atm_cmm_ppa_info`

NAME

`atm_cmm_ppa_info` – Queries parameters from a PPA

SYNOPSIS

```
unsigned long atm_cmm_ppa_info(  
    atm_ppa_p ppa,  
    atm_bind_info_t it);
```

ARGUMENTS

- ppa* Specifies a pointer to the PPA for which information is to be retrieved. This can be any valid PPA currently configured on the system.
- it* Specifies the type of information that is being queried. The value returned depends on the object type of the queried information. You may use the following values to query the indicated information:

Value	Meaning
ATM_PPA_ALLOCGRANE	Returns a type of <code>atm_bw_granularity_p</code> , which reflects the underlying interface's units of bandwidth allocation or its allocation granularity.
ATM_PPA_ALLOCLIMIT	Returns a type of <code>atm_bw_granularity_p</code> , which reflects the underlying interface's per-VC bit rate limits, expressed in allocation granularity units.
ATM_PPA_AVAILRES	Returns a type of <code>atm_services_granes_p</code> , which reflects the amount of bandwidth (in granularity units) currently available for new CBR and pacing circuits over the underlying interface.
ATM_PPA_BRESVLIM	Returns the user-configurable limit, expressed as a percentage of backward bandwidth, on CBR circuits.
ATM_PPA_ESI	Returns the ESI that the signaling module supplied when the PPA was created.
ATM_PPA_ESIID	Returns the unique ESI identifier associated with the PPA's ESI.
ATM_PPA_ESILEN	Returns the length of the ESI that the signaling module supplied when the PPA was created.
ATM_PPA_ESIPID	Returns the ESI identifier of the PPA's parent ESI.

atm_cmm_ppa_info

Value	Meaning
ATM_PPA_FRESVLIM	Returns the user-configurable limit, expressed as a percentage of forward bandwidth, on CBR and pacing circuits.
ATM_PPA_INFO_BURST_AVAIL	Returns the available burst cell rate (in cells-per-second). This is the amount of burst cell rate that is available for a convergence module to reserve.
ATM_PPA_INFO_BURST_MAX	Returns the maximum burst cell rate (in cells-per-second) that the interface supports.
ATM_PPA_INFO_CAPABILITIES	Returns the driver's capabilities for that interface.
ATM_PPA_INFO_DID	Returns a unique global value that can be used to reference the driver for the underlying interface.
ATM_PPA_INFO_DNAME	Returns a pointer to a character string that is the name of the driver for the underlying interface.
ATM_PPA_INFO_DUNIT	Returns the unit number of the driver for the underlying interface.
ATM_PPA_INFO_FC	Returns a nonzero value if hardware flow control is currently enabled on the interface.
ATM_PPA_INFO_HARD_MTU	Returns the largest PDU (in bytes) that the interface supports. This is valid only for interfaces that support AAL5 or AAL3/4 SAR functions in hardware.
ATM_PPA_INFO_HI_VCI	Returns the highest value that may be used for a VCI on the interface.
ATM_PPA_INFO_HI_VPI	Returns the highest value that may be used for a VPI on the interface.
ATM_PPA_INFO_ID	Returns a unique global value for referencing the PPA. Each PPA is assigned a unique ID when it is created. An application program can use this value when specifying a bind point (<code>atmconfig</code> uses these values to reference bind points). The returned value is 32-bits wide.
ATM_PPA_INFO_MAX_VCI	Returns the maximum number of VCIs that the interface supports.
ATM_PPA_INFO_MAX_VPI	Returns the maximum number of VPIs that the interface supports.
ATM_PPA_INFO_MEDIA	Returns a value that indicates the type of physical media to which the interface is connected. The return value is of the type <code>atm_media_type_t</code> .
ATM_PPA_INFO_PEAK_AVAIL	Returns the available peak cell rate (in cells-per-second). This is the amount of peak cell rate that is available for a convergence module to reserve.

atm_cmm_ppa_info

Value	Meaning
ATM_PPA_INFO_PEAK_MAX	Returns the maximum peak cell rate (in cells-per-second) that the interface supports.
ATM_PPA_INFO_QUEUES	Returns the number of scheduling queues that the driver has made visible to the CMM.
ATM_PPA_INFO_SNAME	Returns a pointer to a character string that contains the name of the signaling module that controls all SVCs created on the PPA. If the PPA is a PVC PPA, the returned value is NULL.
ATM_PPA_INFO_SUST_AVAIL	Returns the available sustainable cell rate (in cells-per-second). This is the amount of sustainable cell rate that is available for a convergence module to reserve.
ATM_PPA_INFO_SUST_MAX	Returns the maximum sustainable cell rate (in cells-per-second) that the interface supports.
ATM_PPA_INFO_TOTAL_VC	Returns the maximum number of VCs that can be opened on the interface at any given time.
ATM_PPA_INFO_TYPE	Returns a value that indicates the underlying interface type. The returned value is of the type <code>atm_interface_t</code> .
ATM_PPA_INFO_UNI	Returns the UNI type associated with the PPA, which the signaling module supplied at the time the PPA was created. The return value is of the type <code>atm_uni_type_t</code> .
ATM_PPA_INFO_VC_LEFT	Returns the number of unopened VCs on the interface at the time of the call. This is the total number of VCs (the value returned by <code>ATM_PPA_INFO_TOTAL_VC</code>) minus the number of VCs currently opened.
ATM_PPA_MAX_VC_BBW	Returns the user-configurable maximum per-VC backward (incoming) bit rate (in allocation granularity units) permitted over the underlying interface.
ATM_PPA_MAX_VC_FBW	Returns the user-configurable maximum per-VC forward (outgoing) bit rate (in allocation granularity units) permitted over the underlying interface.
ATM_PPA_MAXRES	Returns a type of <code>atm_services_granes_p</code> , which reflects the maximum bandwidths available for CBR and pacing circuits over the underlying interface. These bandwidths (expressed in granularity units) are a function of the interface limits and the user-configurable limits on the percentage of bandwidth available to CBR and pacing circuits.
ATM_PPA_NUM_VCI	Returns the current number of VCIs configured on the underlying interface.

atm_cmm_ppa_info

Value	Meaning
ATM_PPA_NUM_VPI	Returns the current number of VPIs configured on the underlying interface.
ATM_PPA_PEAK_CELLRATE	Returns the peak PDU bit rate (in cells-per-second) for the underlying interface.
ATM_PPA_UNUSEDRES_BACK	Returns the amount of backward (incoming) bandwidth (in cells-per-second) currently reserved for, but not yet applied to, CBR circuits.
ATM_PPA_UNUSEDRES_FWD	Returns the amount of forward (outgoing) bandwidth (in cells-per-second) currently reserved for, but not yet applied to, CBR and pacing circuits.

DESCRIPTION

The `atm_cmm_ppa_info` routine is a convergence module interface that queries the CMM for current information about an interface's physical capabilities (such as its bit rates) and the amount of specific resources left available on an interface (the remaining nonreserved sustainable bit rate). The CMM and device drivers keep track of this information.

Since convergence modules normally deal with either PPAs or AESA bindings, the CMM allows convergence modules to query the information from the underlying interface by specifying the PPA directly. Thus, the convergence module requires no specific knowledge of any device driver or device driver-related structures.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_bind_info`, `atm_cmm_del_ppa`, `atm_cmm_new_ppa`,
`atm_cmm_ppa_bind`, `atm_cmm_ppa_unbind`

Section 4.4.1.15 for a description of the `capabilities` member of the `atm_drv_params` structure

`atm_cmm_ppa_unbind`

NAME

`atm_cmm_ppa_unbind` – Unbinds convergence module service from a PPA

SYNOPSIS

```
atm_error_t atm_cmm_ppa_unbind(  
    atm_bind_handle_t bind_handle);
```

ARGUMENTS

bind_handle Specifies the binding to be removed. This is the handle that the `atm_cmm_ppa_bind` routine returns to the convergence module.

DESCRIPTION

The `atm_cmm_ppa_unbind` routine is a convergence module interface that unbinds a convergence module's service from a PPA. A convergence module uses this routine when it does not want to accept an incoming connection or to make outgoing calls, for example, a module awaiting a call from a specific caller. Typically, convergence modules should unbind a PPA if their service is no longer available, and must unbind before unregistering with the CMM.

Removing a binding does not affect any currently open VCs.

RETURN VALUES

Upon successful completion, the `atm_cmm_ppa_unbind` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error number.

RELATED INFORMATION

`atm_cmm_bind_info`, `atm_cmm_del_ppa`, `atm_cmm_new_ppa`,
`atm_cmm_ppa_bind`, `atm_cmm_ppa_info`

`atm_cmm_receive`

NAME

`atm_cmm_receive` – Presents received data to the CMM

SYNOPSIS

```
void atm_cmm_receive(  
    atm_drv_handle_t driver,  
    const struct mbuf *const mbp,  
    const int length,  
    const struct mbuf *const trailer,  
    atm_vc_t *vc,  
    char pti,  
    char gfc);
```

ARGUMENTS

- | | |
|----------------|---|
| <i>driver</i> | Specifies the interface on which the data arrived. This is the driver handle that is given to the driver when it registers with the CMM. |
| <i>mbp</i> | Specifies a pointer to a chain of mbufs that contain the data received. |
| <i>length</i> | Specifies the total number of bytes received. For cooked data, this is the number of cells received multiplied by 48. For raw cells, this is the total number of data bytes in the mbuf chain, including the ATM cell headers but not including any time-stamps. |
| <i>trailer</i> | Specifies a pointer to the last mbuf in the mbuf chain. This enables the CMM to quickly locate the cooked packet trailer in the incoming packet without having to traverse the mbuf chain (the driver already knows the address of this mbuf because it must fill in the mbuf length after the data is copied by the adapter using DMA). For raw data, this pointer simply references the last mbuf in the chain. |
| <i>vc</i> | Specifies a pointer to the <code>atm_vc</code> structure of the VC on which the data was received. |

`atm_cmm_receive`

- pti* Specifies the PTI field of the incoming cells. This is driver dependent since it is up to the hardware how it keeps track of the PTI fields of all the cells in a packet. The driver indicates its ability to keep track of these bits to the CMM in the `atm_drv_params` structure during the driver registration process. This field is valid only for cooked connections since complete cells are passed up when raw cells are received.
- gfc* Specifies flow-control information the driver passes up to the CMM. The use of this field is not currently defined. This argument is provided for future use when GFC bits are standardized.

DESCRIPTION

The `atm_cmm_receive` routine is a device driver interface. When an ATM device driver receives data packets and cells from the ATM network, the driver calls this routine to pass the data to the CMM.

When the CMM receives the data, it immediately passes the data to the convergence module that owns the VC. The CMM does not queue the data. That way, the convergence modules receive the data at their input functions in the interrupt context; the driver should do all receive processing in interrupt context for efficiency.

After passing data to the CMM, the driver must not reference any mbufs in the mbuf chain again. If the driver must allocate private storage for data (rather than allocating from the system mbuf pool), the driver must provide an appropriate free routine and set the `m_ext` structure in the mbuf appropriately. The data mbuf chain is deallocated only when the protocol stack has finished referencing the data.

Device drivers use this call to pass up only data packets and cells. Device drivers must use the `atm_cmm_oam_receive` routine to pass operations and maintenance (OAM) cells (nondata cells) to the CMM.

RETURN VALUES

None

`atm_cmm_receive`

RELATED INFORMATION

`xxx_xmit`

Section 3.5.3 for information on the mbuf chain

Section 4.2 for information on receiving cells

atm_cmm_register_cvg

NAME

atm_cmm_register_cvg – Registers a convergence module with the CMM

SYNOPSIS

```
atm_cvg_handle_t atm_cmm_register_cvg(  
    long version,  
    char *const name,  
    void *cvg_handle,  
    void (*const xxx_receive)(atm_vc_p vc,  
        struct mbuf *mdp,  
        int length,  
        struct mbuf *trailer,  
        char pti,  
        char gfc),  
    atm_error_t (*const xxx_except)(void *handle,  
        unsigned int exception,  
        void *arg),  
    int (*const xxx_mmi)(void *handle,  
        unsigned int command,  
        void *arg,  
        int *retval,  
        struct ucred *cred));
```

ARGUMENTS

- | | |
|-------------------|---|
| <i>version</i> | Specifies the version of the include files that were used when the convergence module driver was compiled. The value of this argument must match <code>ATM_REVISION</code> in the <code>/usr/include/io/atm/sys/atm.h</code> file when the convergence module was compiled. This tells the CMM which version of the global data structures the convergence module is using. |
| <i>name</i> | Specifies a unique character string by which the convergence module is known to the system. Management programs use this string to display the name of the convergence module. This string must be unique; if the convergence module registers itself multiple times, it must use a different name for each registration. |
| <i>cvg_handle</i> | Specifies a value that the convergence module can use to identify a specific registration to the CMM. |

`atm_cmm_register_cvg`

This value is passed back to the convergence module for exception and management processing. The CMM does not modify this value, which is always passed back unchanged.

`xxx_receive`

Specifies a pointer to a routine that CMM uses to notify the convergence module of incoming data. A convergence module must supply a receive routine.

`xxx_except`

Specifies a pointer to a routine that CMM uses to notify the convergence module of new connections and exception conditions. A convergence module must supply an exception routine.

`xxx_mmi`

Specifies a pointer to a routine that CMM uses to manage the convergence module through the ATM Module Management Interface (MMI). A convergence module must supply an `xxx_mmi` routine if it is to be managed through the MMI.

DESCRIPTION

The `atm_cmm_register_cvg` routine is a convergence module routine that registers a convergence module with the CMM. A protocol convergence module must register with the CMM before it can interact with the ATM subsystem. The CMM passes interface configuration information (for example, currently configured PPAs) to the convergence module, using the module's exception routine, before the registration call returns.

Once a convergence module is registered, the CMM knows where to deliver connection notifications and various exception notifications (for example, changes in system configuration). Convergence modules can register at any time.

RETURN VALUES

The `atm_cmm_register_cvg` routine returns a unique value that identifies an instance of the convergence module. The convergence module must use this value on all subsequent calls to the CMM.

`atm_cmm_register_cvg`

RELATED INFORMATION

`atm_cmm_unregister_cvg`, `xxx_except`, `xxx_mmi`, `xxx_receive`

Chapter 8 for information on the MMI

atm_cmm_register_dd

NAME

atm_cmm_register_dd – Registers a device driver with the CMM

SYNOPSIS

```
atm_drv_handle_p atm_cmm_register_dd(  
    long version,  
    atm_drv_params_p params,  
    atm_error_t (*const xxx_xmit)(int unit,  
        struct mbuf *data,  
        long length,  
        atm_vc_p vc,  
        unsigned char clp,  
        unsigned char gfc),  
    atm_error_t (*const xxx_manage)(int unit,  
        unsigned int command,  
        void *arg),  
    int (*const xxx_mmi)(int handle,  
        unsigned int command,  
        void *arg,  
        int *retval,  
        struct ucred *cred));
```

ARGUMENTS

<i>version</i>	Specifies the version of the include files that are used when the device driver is compiled. The value of this argument must match <code>ATM_REVISION</code> in the <code>/usr/include/io/atm/sys/atm.h</code> file when the device driver is compiled. This tells the CMM which version of the global data structures the driver is using.
<i>params</i>	Specifies a pointer to an <code>atm_drv_params</code> structure, which defines the basic capabilities of the interface. This structure must be completely initialized. For the registration call, this pointer can reference static memory since the CMM copies the information into its local memory before returning.
<i>xxx_xmit</i>	Specifies the address of the routine for the CMM to call when it has data to transmit on the interface.

atm_cmm_register_dd

<i>xxx_manage</i>	Specifies the address of a routine that the CMM calls to perform various driver management functions. The management function definitions are in the <code>atm_adi.h</code> file in the <code>/usr/include/io/atm/sys</code> directory.
<i>xxx_mmi</i>	Specifies a pointer to a routine that the CMM calls to pass MMI commands from an application program. This routine is optional; you provide it only if the driver implements any MMI services.

DESCRIPTION

The `atm_cmm_register_dd` routine is a device driver interface that registers the device driver with the CMM. This is done during driver initialization, when a device driver's attach routine is called. The `atm_cmm_register_dd` routine is called before returning from the attach.

RETURN VALUES

Upon successful completion, `atm_cmm_register_dd` returns `atm_drv_handle_t`. The driver uses this handle in all subsequent calls to the CMM; otherwise, it returns `NULL`.

RELATED INFORMATION

`xxx_manage`, `xxx_mmi`, `xxx_xmit`

Section 3.5.3 for information on the mbuf chain

Chapter 4 for more information on the device driver interface and the `atm_queue_param` structure

Chapter 8 for more information on the MMI

`atm_cmm_register_sig`

NAME

`atm_cmm_register_sig` – Registers the signaling module with the CMM

SYNOPSIS

```
atm_sig_handle_t atm_cmm_register_sig(  
    long version,  
    char *const id,  
    void *sig_handle,  
    atm_error_t (*const xxx_setup)(atm_addr_p addr,  
        unsigned long *refptr),  
    atm_error_t (*const xxx_release)(atm_addr_p addr),  
    atm_error_t (*const xxx_add)(atm_addr_p addr),  
    atm_error_t (*const xxx_drop)(atm_addr_p addr),  
    atm_error_t (*const xxx_enquery)(atm_addr_p addr),  
    atm_error_t (*const xxx_restart)(void *handle,  
        unsigned int class,  
        unsigned int vpi,  
        unsigned int vci),  
    atm_error_t (*const xxx_except)(void *handle,  
        unsigned int exception,  
        void *arg),  
    int (*const xxx_mmi)(void *handle,  
        unsigned int command,  
        void *arg,  
        int *retval,  
        struct ucred *cred),  
    atm_error_t (*const xxx_mib)(void *sig_handle,  
        unsigned int command,  
        atm_oid_p oid,  
        unsigned int fieldlen,  
        unsigned char *fieldval,  
        void *id));
```

ARGUMENTS

- | | |
|----------------|--|
| <i>version</i> | Specifies the version of the include files that were used when the signaling module was compiled. The value of this argument must match <code>ATM_REVISION</code> in the <code>/usr/include/io/atm/sys/atm.h</code> file when the signaling module was compiled. This tells the CMM which version of the global data structures the signaling module is using. |
| <i>id</i> | Specifies a unique character string by which the protocol is known to the system. This string is used both by management programs to display the name |

atm_cmm_register_sig

of the switching module and by protocol convergence modules needing to use a specific protocol. For example, the UNI 3.0 switching protocol could use an ID of UNI 3.0 to identify itself. Each ID must be unique. The CMM performs signaling module lookup with case-insensitive searches, so the IDs cannot use case to differentiate themselves. Management programs display the ID strings as supplied by the signaling modules. Signaling modules that implement more than one protocol (or more than one version of the same protocol) must use multiple registration calls to register each protocol.

<i>sig_handle</i>	Specifies a value that the signaling module can use to identify a specific registration to the CMM. This value is passed back to the signaling module for exception and management processing. The CMM does not modify this value, which is always passed back unchanged.
<i>xxx_setup</i>	Specifies a pointer to a routine that the CMM calls to request the creation of a new connection (make a call).
<i>xxx_release</i>	Specifies a pointer to a routine that the CMM calls to request the deletion of a connection (a hangup).
<i>xxx_add</i>	Specifies a pointer to a routine that the CMM calls to request the addition of an endpoint to a point-to-multipoint connection.
<i>xxx_drop</i>	Specifies a pointer to a routine that the CMM calls to request the dropping of an endpoint to a point-to-multipoint connection.
<i>xxx_enquery</i>	Specifies a pointer to a routine that the CMM calls to request status for an endpoint.

atm_cmm_register_sig

<i>xxx_restart</i>	Specifies a pointer to a routine that the CMM calls to request a RESTART message be sent.
<i>xxx_except</i>	Specifies a pointer to a routine that CMM uses to notify the signaling module of errors and exception conditions.
<i>xxx_mmi</i>	Specifies a pointer to a routine that CMM uses to manage the convergence module through the ATM Module Management Interface (MMI). A convergence module must supply an <i>xxx_mmi</i> routine if it is to be managed through the MMI.
<i>xxx_mib</i>	Specifies a pointer to a routine that CMM uses to access fields of MIBs that the signaling modules maintain. You provide this routine only when a signaling module maintains a MIB.

DESCRIPTION

The `atm_cmm_register_sig` routine is a signaling module interface that registers a signaling module with the CMM. Registration must occur when the signaling module is initialized. Once registered, the CMM can use the signaling module and make it available to convergence modules.

Signaling modules should register as both signaling and convergence modules. When registering as a convergence module, a signaling module must set up its signaling VCs to send and receive signaling messages. The CMM does not provide special facilities for signaling VCs, but treats them like any other VC.

RETURN VALUES

None

RELATED INFORMATION

`xxx_add`, `xxx_drop`, `xxx_enquery`, `xxx_except`, `xxx_mib`, `xxx_mmi`,
`xxx_release`, `xxx_restart`, `xxx_setup`

`atm_cmm_register_sig`

Section 7.6 for a description of signaling VC initialization

Chapter 8 for a discussion of the MMI

`atm_cmm_reject`

NAME

`atm_cmm_reject` – Rejects an incoming call and notifies the CMM

SYNOPSIS

```
void atm_cmm_reject(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure for the call being rejected. This is the same pointer passed into the convergence module's `xxx_connect` routine when the call first came in.

DESCRIPTION

The `atm_cmm_reject` routine is a convergence module interface that rejects a previously deferred incoming call and notifies the CMM. The incoming call was deferred when a convergence module returned `ATM_CAUSE_DEFER` from its `xxx_connect` routine.

When this call returns, the convergence module must destroy all references to the rejected call.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_accept`, `atm_cmm_new_call`

atm_cmm_release

NAME

`atm_cmm_release` – Requests that a VC be disconnected

SYNOPSIS

```
atm_err_t atm_cmm_release(  
    atm_cvg_handle_t cm,  
    atm_vc_t *vc);
```

ARGUMENTS

cm Specifies the value that the registration routine returns to the convergence module. This uniquely identifies the convergence module making the request.

vc Specifies a pointer to the VC to be torn down. This can be a point-to-point or a point-to-multipoint VC. In the case of a point-to-multipoint VC, connections to all endpoints in the multipoint connection are torn down.

DESCRIPTION

The `atm_cmm_release` routine is a convergence module routine that requests the CMM to delete a virtual circuit. A convergence module uses this routine when it no longer requires a connection to a remote host. After the routine is called, release negotiation with the remote host is initiated, which will eventually lead to the connection being torn down and all resources released.

A convergence module can either request that a specific endpoint be dropped using `atm_cmm_drop` or that a VC be disconnected. These operations are equivalent on point-to-point connections; disconnecting only the endpoint results in the VC being torn down. If a convergence module requests that a point-to-multipoint VC be disconnected, the CMM first disconnects all endpoints associated with the VC. This task simplifies the task of disconnecting multipoint VCs and for disconnecting PVCs.

RETURN VALUES

For an SVC, if the connection teardown is proceeding, the `atm_cmm_release` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns

`atm_cmm_release`

an ATM error number. As each endpoint is torn down, the convergence module receives an `ATM_CME_EP_DEAD` exception notification for that endpoint. When the last endpoint associated with a VC has been disconnected, the VC is torn down.

For a PVC, if the PVC has been destroyed on the local system, the `atm_cmm_release` routine returns `ATM_CAUSE_GOOD` and the PVC can still exist on the network; otherwise, it returns an ATM error number. No notifications are provided on the PVC release.

RELATED INFORMATION

`atm_cmm_con_release`

`atm_cmm_reply`

NAME

`atm_cmm_reply` – Notifies the CMM that an outgoing connection has been acknowledged

SYNOPSIS

```
atm_err_t atm_cmm_reply(
    atm_sig_handle_t sm,
    atm_addr_t *addr,
    atm_uni_call_ie_p ies,
    unsigned int vpi,
    unsigned int vci,
    unsigned long call_reference);
```

ARGUMENTS

- | | |
|-------------|--|
| <i>sm</i> | Specifies the signaling module's handle assigned to it when it registered with the CMM. |
| <i>addr</i> | Specifies a pointer to an <code>atm_addr</code> structure that contains all the parameters of the call. This pointer must point to the same <code>atm_addr</code> structure that was passed in the <code>xxx_setup</code> and <code>xxx_add</code> routines since this pointer is the handle used to refer to the specific connection. |
| <i>ies</i> | Specifies a pointer to an array of UNI signaling information elements (or any memory array in the case of non-UNI signaling protocols). The signaling module stores the contents of the information elements sent in the connection reply received from the called party in the array. The information element array contains only those elements that do not translate to values in the <code>atm_vc_services</code> structure. |
| <i>vpi</i> | Specifies the VPI value for the connection. The network should assign this before the routine is called. |

atm_cmm_reply

<i>vci</i>	Specifies the VCI value for the connection. The network should assign this before the routine is called.
<i>call_reference</i>	Specifies the call reference value that the signaling module will use when referring to this call. This is placed in the <code>atm_vc</code> structure for future use in referencing the call.

DESCRIPTION

The `atm_cmm_reply` routine is a signaling module interface. Signaling modules use this routine to notify the CMM when they receive notification from the switch that an outgoing connection has been completed. The CMM receives signaling information elements returned by the called party in the connection reply. You use this call to notify the CMM of both the completion of point-to-point connections and of the addition of endpoints in point-to-multipoint connections.

If you are adding endpoints to a point-to-multipoint connection, the `addr` pointer will point to the `atm_addr` structure of the added endpoint.

The call to `atm_cmm_reply` is always followed by a call to `atm_cmm_activate_con`, which notifies the CMM that the connection is ready to carry data.

RETURN VALUES

Since the `atm_cmm_reply` routine is notifying the CMM that a previously created connection has completed, it can fail only if the connection referenced is invalid. In this case, it returns `ATM_CAUSE_EIR` to indicate it has no previous entry for the connection; otherwise, it returns `ATM_CAUSE_GOOD`. If the CMM or a convergence module needs to destroy the connection at this point it must follow the normal teardown procedure. If `ATM_CAUSE_EIR` is returned, the signaling module is expected to tear down the connection.

RELATED INFORMATION

`atm_cmm_activate_con`

`atm_cmm_reserve_resources`

NAME

`atm_cmm_reserve_resources` – Reserves resources for CBR and pacing services

SYNOPSIS

```
atm_err_t atm_cmm_reserve_resources(  
    atm_ppa_p ppa,  
    atm_cvg_handle_t cm,  
    atm_vc_services_p s);
```

ARGUMENTS

- | | |
|------------|--|
| <i>ppa</i> | Specifies the underlying driver or adapter bandwidth pool from which to allocate the resources. The underlying driver or adapter must support CBR or pacing. These resources can be applied only to point-to-point connections set up over bind points associated with the same PPA. |
| <i>cm</i> | Specifies the value returned to the convergence module by the registration request routine. This uniquely identifies the convergence module reserving the resources. |
| <i>s</i> | Specifies an <code>atm_vc_services_p</code> pointing to a properly set <code>atm_vc_services</code> structure that defines the CBR or pacing circuit parameters. |

DESCRIPTION

The `atm_cmm_reserve_resources` routine is a convergence module interface. Convergence modules use this routine to reserve bandwidth resources and to notify the CMM before making a call or accepting a connection. The CMM passes the notification to the device driver and protocol convergence module.

RETURN VALUES

The `atm_cmm_reserve_resources` routine returns `ATM_CAUSE_GOOD`, if the CMM was able to allocate and reserve the necessary resources.

atm_cmm_reserve_resources

This routine returns `ATM_CAUSE_DEFER` if the CMM was not able to allocate and reserve the necessary resources. In this case, the convergence module must wait for the `ATM_CME_RESV_AVAIL` notification exception before making a call or accepting a connection. Until the convergence module receives an `ATM_CME_RESV_AVAIL` notification exception, the `atm_vc_services` structure is considered to be a resource reservation request.

If the resource reservation fails, the `atm_cmm_reserve_resources` routine returns an ATM error number that indicates the reason for the resource reservation failure. If the failure is due to an incorrectly set `atm_vc_services` structure, the `lerrstat` structure member contains local error information.

RELATED INFORMATION

`atm_cmm_free_services`

`atm_cmm_restart`

NAME

`atm_cmm_restart` – Notifies the CMM that a RESTART request was received

SYNOPSIS

```
atm_err_t atm_cmm_restart(  
    atm_sig_handle_t sm,  
    unsigned int class,  
    unsigned int vpi,  
    unsigned int vci);
```

ARGUMENTS

- sm* Specifies the signaling module's handle returned by the CMM when the signaling module registers.
- class* Specifies the type of restart that is being performed. The caller can set the value of this argument to either `ATM_RESTART_VP` (to restart all VCs on a VP) or `ATM_RESTART_VC` (to restart a specific VC).
- vpi* Specifies the VC(s) to be restarted. If *class* is set to `ATM_RESTART_VP`, *vpi* specifies the VP on which all VCs are to be restarted; the *vci* argument is ignored. If *class* is set to `ATM_RESTART_VC`, the *vpi* and *vci* arguments specify which VC is to be restarted.
- vci* Specifies the VC(s) to be restarted. If *class* is set to `ATM_RESTART_VC`, the *vpi* and *vci* arguments specify which VC is to be restarted.

DESCRIPTION

The `atm_cmm_restart` routine is a signaling module interface. When a signaling module receives a RESTART request, the module calls this routine to pass all of the appropriate information to the CMM. The CMM then initiates a restart of the indicated VC(s) before returning to the signaling module.

`atm_cmm_restart`

RETURN VALUES

If the restart is successfully initiated, the `atm_cmm_restart` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an appropriate ATM error number.

RELATED INFORMATION

`atm_cmm_new_call`, `atm_cmm_restart_ack`

`atm_cmm_restart_ack`

NAME

`atm_cmm_restart_ack` – Notifies the CMM that a restart has completed

SYNOPSIS

```
void atm_cmm_restart_ack(  
    void *handle,  
    unsigned int class,  
    unsigned int vpi,  
    unsigned int vci);
```

ARGUMENTS

- handle* Specifies the handle that the CMM uses to identify the restart. This value has significance only to the CMM. This is the value that the CMM passed to the restart function call.
- class* Specifies the type of restart that is being performed. The caller can set the value of this argument to either `ATM_RESTART_VP` (to restart all VCs on a VP) or `ATM_RESTART_VC` (to restart a specific VC).
- vpi* Specifies the VC(s) to be restarted. If *class* is set to `ATM_RESTART_VP`, *vpi* specifies the VP on which all VCs are to be restarted; the *vci* argument is ignored. If *class* is set to `ATM_RESTART_VC`, the *vpi* and *vci* arguments specify which VC is to be restarted.
- vci* Specifies the VC(s) to be restarted. If *class* is set to `ATM_RESTART_VC`, the *vpi* and *vci* arguments specify which VC is to be restarted.

DESCRIPTION

The `atm_cmm_restart_ack` routine is a signaling module interface that signaling modules call to notify the CMM when a restart has been completed. Once notified, the CMM returns the indicated VCs to the NULL state and frees their resources.

`atm_cmm_restart_ack`

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_restart`

`atm_cmm_send`

NAME

`atm_cmm_send` – Transmits data on an established VC

SYNOPSIS

```
atm_err_t atm_cmm_send(  
    atm_cvg_handle_t cm,  
    atm_vc_t *vc,  
    const struct mbuf *const data,  
    const long length,  
    unsigned char clp,  
    unsigned char gfc);
```

ARGUMENTS

- | | |
|---------------|--|
| <i>cm</i> | Specifies the value returned to the convergence module by the registration function call. This uniquely identifies the convergence module making the request. |
| <i>vc</i> | Specifies a pointer to the <code>atm_vc</code> structure of the VC on which the data is to be sent. |
| <i>data</i> | Specifies a pointer to a chain of mbufs that contain the data to be transmitted. |
| <i>length</i> | Specifies the total length of the data to be sent. For raw ATM cell transmission, this value must be a multiple of the ATM cell size. For cooked transmission, this value indicates the size of the PDU to be encapsulated and sent. |
| <i>clp</i> | Specifies the congestion loss priority bits to be used when transmitting cooked data. When transmitting raw data, the convergence module sets these bits in each ATM cell. |
| <i>gfc</i> | Specifies the value of the generic flow-control bits to be used when transmitting the cooked data. When transmitting raw data, the convergence module sets these bits in each ATM cell. |

`atm_cmm_send`

DESCRIPTION

The `atm_cmm_send` routine is a convergence module interface that transmits data on an established virtual circuit. The CMM then hands the data to the appropriate device driver. Although the CMM does not queue outgoing data, the device driver might. Therefore, the successful return from this routine does not imply that the data was actually transmitted. In fact, the data could be discarded in the driver or on the network. The convergence module is not notified if the data is dropped locally.

RETURN VALUES

If the data is successfully passed to the driver and the driver accepts the data, the `atm_cmm_send` routine returns `ATM_CAUSE_GOOD`. This does not indicate that the data arrived at the destination or that it has even left the system. If the driver does not accept the data, an ATM error number is returned, indicating the reason the data was not accepted.

`ATM_CAUSE_QFULL` indicates that the driver cannot accept the data because its queue is full or it is out of some other resource. The convergence module can then decide if the data should be discarded or queued to send later.

RELATED INFORMATION

`xxx_xmit`

Section 3.5.3 for more information on the mbuf chain

Chapter 10 for information on flow control in the ATM subsystem

`atm_cmm_set_cause`

NAME

`atm_cmm_set_cause` – Logs a network-visible VC or endpoint condition

SYNOPSIS

```
atm_err_t atm_cmm_set_cause(  
    atm_cvg_handle_t cm,  
    atm_vc_p vp,  
    atm_addr_p addr,  
    char *reason,  
    atm_error_t cause,  
    atm_location_t location,  
    unsigned char diag_length,  
    unsigned char *diag);
```

ARGUMENTS

<i>cm</i>	Specifies the value the registration request function call returned to the convergence module. This uniquely identifies the convergence module making the request.
<i>vp</i>	Specifies a pointer to the VC on which the condition occurred. If the error applies to one endpoint, set the value to NULL and set the <i>addr</i> value to non-NULL.
<i>addr</i>	Specifies a pointer to the <code>atm_addr</code> structure for the endpoint on which the error occurred. If the error applies to all endpoints on the VC (global), set the value to NULL and set the <i>vp</i> value to non-NULL.
<i>reason</i>	Specifies a NULL-terminated character string that contains descriptive text for the error.
<i>cause</i>	Specifies the <code>atm_error_t</code> value that describes the condition.

atm_cmm_set_cause

<i>location</i>	Specifies an <code>atm_location_t</code> value that identifies the location in the network where the error occurred.
<i>diag_length</i>	Specifies the length (in bytes) of the data in the <i>diag</i> argument.
<i>diag</i>	Specifies a string of bytes that contains diagnostic information about the error. This data might be passed to the network in the <code>diag</code> field of a Cause IE.

DESCRIPTION

The `atm_cmm_set_cause` routine is a convergence module interface that records an error or normal condition associated with a VC or endpoint. The condition is stored as an `atm_cause_info` structure together with the VC. Convergence and signaling modules can retrieve this information by using the `atm_cmm_next_cause` routine. Users can display this information by using the `atmconfig` utility.

Errors or events stored with the `atm_cmm_set_cause` routine are visible to the network and to the other end of the connection. When a signaling module releases a connection or drops an endpoint, the module extracts the most recent cause from a VC or endpoint and creates a Cause IE.

RETURN VALUES

If the cause is recorded successfully, the `atm_cmm_set_cause` routine returns `ATM_CAUSE_GOOD`. If the driver handle, *vp* argument, or *addr* argument is invalid, the routine returns `ATM_CAUSE_BARG`.

RELATED INFORMATION

`atm_cmm_adi_set_cause`, `atm_cmm_next_cause`, `atm_cmm_set_log`,
`atm_cmm_smi_set_cause`

Section 3.10.7 for information on cause information

`atm_cmm_set_log`

NAME

`atm_cmm_set_log` – Logs a VC or endpoint condition

SYNOPSIS

```
atm_err_t atm_cmm_set_log(  
    atm_cvg_handle_t cm,  
    atm_vc_p vp,  
    atm_addr_p addr,  
    char *reason,  
    atm_error_t cause,  
    atm_location_t location,  
    unsigned char diag_length,  
    unsigned char *diag);
```

ARGUMENTS

<i>cm</i>	Specifies the value the registration request function call returned to the convergence module. This uniquely identifies the convergence module making the request.
<i>vp</i>	Specifies a pointer to the VC on which the condition occurred. If the error applies to one endpoint, set the value to NULL and set the <i>addr</i> value to non-NULL.
<i>addr</i>	Specifies a pointer to the <code>atm_addr</code> structure for the endpoint on which the error occurred. If the error applies to all endpoints on the VC (global), set the value to NULL and set the <i>vp</i> value to non-NULL.
<i>reason</i>	Specifies a NULL-terminated character string that contains descriptive text for the error.
<i>cause</i>	Specifies the <code>atm_error_t</code> value that describes the condition.

`atm_cmm_set_log`

<i>location</i>	Specifies an <code>atm_location_t</code> value that identifies the location in the network where the error occurred.
<i>diag_length</i>	Specifies the length (in bytes) of the data in the <i>diag</i> argument.
<i>diag</i>	Specifies a string of bytes that contains diagnostic information about the error. This data might be passed to the network in the <code>diag</code> field of a Cause IE.

DESCRIPTION

The `atm_cmm_set_log` routine is a convergence module interface that records an error or normal condition associated with a VC or endpoint. The condition is stored as an `atm_cause_info` structure together with the VC. Users can display this information by using the `atmconfig` utility.

Errors or events stored with the `atm_cmm_set_log` routine are logged on the local system only, and are not available when a signaling module generates a Cause IE. Logging provides information about VC or endpoint activity for a system or network administrator to view on the local system.

RETURN VALUES

If the cause is recorded successfully, the `atm_cmm_set_log` routine returns `ATM_CAUSE_GOOD`. If the `handle`, `vp` argument, or `addr` argument is invalid, the routine returns `ATM_CAUSE_BARG`.

RELATED INFORMATION

`atm_cmm_adi_set_log`, `atm_cmm_next_cause`, `atm_cmm_set_cause`,
`atm_cmm_smi_set_log`

Section 3.10.7 for information on cause information

atm_cmm_smi_set_cause

NAME

atm_cmm_smi_set_cause – Logs a network-visible VC or endpoint condition

SYNOPSIS

```
atm_err_t atm_cmm_smi_set_cause(  
    atm_sig_handle_t sm,  
    atm_vc_p vp,  
    atm_addr_p addr,  
    char *reason,  
    atm_error_t cause,  
    atm_location_t location,  
    unsigned char diag_length,  
    unsigned char *diag);
```

ARGUMENTS

<i>sm</i>	Specifies the signaling module handle that the registration function call returned.
<i>vp</i>	Specifies a pointer to the VC on which the condition occurred. If the error applies to one endpoint, set the value to NULL and set the <i>addr</i> value to non-NULL.
<i>addr</i>	Specifies a pointer to the <i>atm_addr</i> structure for the endpoint on which the error occurred. If the error applies to all endpoints on the VC (global), set the value to NULL and set the <i>vp</i> value to non-NULL.
<i>reason</i>	Specifies a NULL-terminated character string that contains descriptive text for the error.
<i>cause</i>	Specifies the <i>atm_error_t</i> value that describes the condition.
<i>location</i>	Specifies an <i>atm_location_t</i> value that identifies the location in the network where the error occurred.

atm_cmm_smi_set_cause

<i>diag_length</i>	Specifies the length (in bytes) of the data in the <i>diag</i> argument.
<i>diag</i>	Specifies a string of bytes that contains diagnostic information about the error. This data might be passed to the network in the <i>diag</i> field of a Cause IE.

DESCRIPTION

The `atm_cmm_smi_set_cause` routine is a signaling module interface that records an error or normal condition associated with a VC or endpoint. The condition is stored as an `atm_cause_info` structure together with the VC. Convergence and signaling modules can retrieve this information by using the `atm_cmm_next_cause` routine. Users can display this information by using the `atmconfig` utility.

Errors or events stored with the `atm_cmm_smi_set_cause` routine are visible to the network and to the other end of the connection. When a signaling module releases a connection or drops an endpoint, the module extracts the most recent cause from a VC or endpoint and creates a Cause IE.

RETURN VALUES

If the cause is recorded successfully, the `atm_cmm_smi_set_cause` routine returns `ATM_CAUSE_GOOD`. If the driver `handle`, `vp` argument, or `addr` argument is invalid, the routine returns `ATM_CAUSE_BARG`.

RELATED INFORMATION

`atm_cmm_adi_set_cause`, `atm_cmm_smi_set_log`,
`atm_cmm_next_cause`, `atm_cmm_set_cause`

Section 3.10.7 for information on cause information

atm_cmm_smi_set_log

NAME

atm_cmm_smi_set_log – Logs a VC or endpoint condition

SYNOPSIS

```
atm_err_t atm_cmm_smi_set_log(  
    atm_sig_handle_t sm,  
    atm_vc_p vp,  
    atm_addr_p addr,  
    char *reason,  
    atm_error_t cause,  
    atm_location_t location,  
    unsigned char diag_length,  
    unsigned char *diag);
```

ARGUMENTS

<i>sm</i>	Specifies the signaling module handle that the registration function call returned.
<i>vp</i>	Specifies a pointer to the VC on which the condition occurred. If the error applies to one endpoint, set the value to NULL and set the <i>addr</i> value to non-NULL.
<i>addr</i>	Specifies a pointer to the <i>atm_addr</i> structure for the endpoint on which the error occurred. If the error applies to all endpoints on the VC (global), set the value to NULL and set the <i>vp</i> value to non-NULL.
<i>reason</i>	Specifies a NULL-terminated character string that contains descriptive text for the error.
<i>cause</i>	Specifies the <i>atm_error_t</i> value that describes the condition.
<i>location</i>	Specifies an <i>atm_location_t</i> value that identifies the location in the network where the error occurred.

`atm_cmm_smi_set_log`

<i>diag_length</i>	Specifies the length (in bytes) of the data in the <i>diag</i> argument.
<i>diag</i>	Specifies a string of bytes that contains diagnostic information about the error. This data might be passed to the network in the <i>diag</i> field of a Cause IE.

DESCRIPTION

The `atm_cmm_smi_set_log` routine is a signaling module interface that records an error or normal condition associated with a VC or endpoint. The condition is stored as an `atm_cause_info` structure together with the VC. Users can display this information by using the `atmconfig` utility.

Errors or events stored with the `atm_cmm_smi_set_log` routine are logged on the local system only, and are not available when a signaling module generates a Cause IE. Logging provides information about VC or endpoint activity for a system or network administrator to view on the local system.

RETURN VALUES

If the cause is recorded successfully, the `atm_cmm_smi_set_log` routine returns `ATM_CAUSE_GOOD`. If the `handle`, `vp` argument, or `addr` argument is invalid, the routine returns `ATM_CAUSE_BARG`.

RELATED INFORMATION

`atm_cmm_adi_set_log`, `atm_cmm_next_cause`, `atm_cmm_set_log`,
`atm_cmm_smi_set_cause`

Section 3.10.7 for information on cause information

`atm_cmm_status_done`

NAME

`atm_cmm_status_done` – Notifies the CMM that a status enquiry has completed

SYNOPSIS

```
void atm_cmm_status_done(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure, indicating the endpoint from which the enquiry response was received. This should correspond to the endpoint provided to the signaling module's enquiry function call.

DESCRIPTION

The `atm_cmm_status_done` routine is a signaling module interface that signaling modules call to notify the CMM when a status enquiry has completed. Once notified, the CMM either examines the enquiry data itself or passes it to the convergence module that requested the enquiry.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_enquery`

`atm_cmm_unregister_cvg`

NAME

`atm_cmm_unregister_cvg` – Unregisters a convergence module from the CMM

SYNOPSIS

```
atm_error_t atm_cmm_unregister_cvg(  
    atm_cvg_handle_t cm);
```

ARGUMENTS

cm Specifies a value that identifies the convergence module. The `atm_cmm_register_cvg` call returns this value.

DESCRIPTION

The `atm_cmm_unregister_cvg` routine is a convergence module interface that unregisters a convergence module with the CMM. A convergence module unregisters with the CMM if it no longer needs to receive or make connections. This might be necessary in environments when protocol stacks are dynamically configured and unconfigured from the system.

Before unregistering, a convergence module must close all existing connections and unbind from all PPAs. If all connections are not closed or if PPA bindings still exist, the CMM unregisters only the convergence module's connection routine. The CMM will continue to call the convergence module with incoming data and exception notifications. Once the convergence module is unregistered, the CMM does not call any of its routines.

RETURN VALUES

If the convergence module is completely unregistered and the CMM will make no more calls to its routines, the `atm_cmm_unregister_cvg` routine returns `ATM_CAUSE_GOOD`. If there are still connections associated with the convergence module, it returns `ATM_CAUSE_BUSY`. This means the module's `xxx_connect` routine is unregistered, but the CMM will continue to deliver data and exception notifications. Once the convergence module unregisters, the CMM will not accept any more connection requests from

`atm_cmm_unregister_cvg`

the module until it reregisters. The convergence module must make the unregister call again once all existing connections are closed.

RELATED INFORMATION

`atm_cmm_register_cvg`

`atm_cmm_vc_control`

NAME

`atm_cmm_vc_control` – Modifies VC parameters

SYNOPSIS

```
atm_err_t atm_cmm_vc_control(  
    atm_cvg_handle_t cm,  
    atm_vc_t *vc,  
    int operation,  
    void *arg);
```

ARGUMENTS

<i>cm</i>	Specifies the value returned to the convergence module by the registration function call. This uniquely identifies the convergence module making the request.
<i>vc</i>	Specifies the VC to which the function call applies.
<i>operation</i>	Specifies the type of operation to be performed on the VC.
<i>arg</i>	Specifies further data for the operations. The type and use of this argument depends on the type of operation being performed. The following are valid operations:

atm_cmm_vc_control

Operation	Meaning
ATM_VCC_AGING	Changes the circuit-aging algorithm applied to the VC. For this operation, <i>arg</i> is an <i>int</i> whose value is the new aging parameter (see Table 7-1 for a list of values).
ATM_VCC_QPARAM	Sets the queuing parameters for queuing raw cells in the driver. The argument to this operation is an <i>atm_queue_param_p</i> that points to a properly set <i>atm_queue_param</i> structure. This structure is referenced only within the context of the function call and may be a local variable in the calling function.

DESCRIPTION

The *atm_cmm_vc_control* routine is a convergence module interface that enables convergence modules to modify virtual circuit parameters, either to make the transfer of data more efficient or to control the VC aging. The QOS and circuit bandwidth parameters cannot be modified.

RETURN VALUES

Upon successful completion, the *atm_cmm_vc_control* routine returns *ATM_CAUSE_GOOD*; otherwise, it returns an ATM error number.

RELATED INFORMATION

Section 4.4.2 for information on the *atm_queue_param* structure

Section 7.3 for information on connection aging

`atm_cmm_vc_get`

NAME

`atm_cmm_vc_get` – Requests status information from a VC

SYNOPSIS

```
atm_vc_p atm_cmm_vc_get(  
    atm_drv_handle_t driver,  
    unsigned int vpi,  
    unsigned int vci);
```

ARGUMENTS

- | | |
|---------------|---|
| <i>driver</i> | Specifies the device driver handle for the PPA to which the VC being queried is attached. |
| <i>vpi</i> | Specifies the VPI values for the VC. |
| <i>vci</i> | Specifies the VCI values for the VC. |

DESCRIPTION

The `atm_cmm_vc_get` routine is a signaling module interface. When a signaling module needs to access status information about a VC that is currently in service, based on the VPI and VCI, the module calls this routine.

RETURN VALUES

If the VC is currently configured on the system, the `atm_cmm_vc_get` routine returns a pointer to that `atm_vc` structure; otherwise, it returns NULL.

RELATED INFORMATION

`atm_cmm_status_done`, `atm_cmm_vc_stats`

`atm_cmm_vc_stats`

NAME

`atm_cmm_vc_stats` – Retrieves current VC information from the CMM

SYNOPSIS

```
atm_error_t atm_cmm_vc_stats(  
    atm_vc_p vc,  
    atm_vc_stats_p stats);
```

ARGUMENTS

- vc* Specifies a pointer to the VC for which statistics are to be reported.
- stats* Specifies a pointer to an `atm_vc_stats` structure allocated by the caller into which the CMM can copy the statistics.

DESCRIPTION

The `atm_cmm_vc_stats` routine is a convergence module interface that retrieves VC statistics from the CMM. The CMM returns the information in the `atm_vc_stats` structure.

The CMM keeps track of VC usage on a per-VC basis. This is done as part of the CMM's VC management and cannot be disabled.

No entity can reset the VC counters. If a convergence module needs to collect statistics between two events, it must query the statistics at the first event and the second event and subtract the first event's statistics from the second event's statistics.

RETURN VALUES

Upon successful completion, the `atm_cmm_vc_stats` routine returns `ATM_CAUSE_GOOD`; otherwise, it returns an ATM error.

RELATED INFORMATION

`atm_cmm_vc_control`, `atm_cmm_vc_get`

`xxx_add`

NAME

`xxx_add` – Requests a signaling module to add an endpoint to a point-to-multipoint connection

SYNOPSIS

```
atm_error_t xxx_add(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure for the endpoint to be added.

DESCRIPTION

The `xxx_add` routine is a routine declared within a signaling module that the CMM calls when it needs to add an endpoint to a point-to-multipoint connection. The name of the routine (`xxx_add`) can be any valid C language function name. You pass the routine to the CMM by reference only.

The signaling module can obtain the information about the root's VC by following the `addr->vc` pointer. This routine is nonblocking. The CMM is notified when the connection has been added.

RETURN VALUES

The `xxx_add` routine must return `ATM_CAUSE_GOOD` if the signaling module is proceeding with the endpoint addition or an ATM error number if the call cannot proceed.

RELATED INFORMATION

`atm_cmm_register_sig`

xxx_connect

NAME

`xxx_connect` – Notifies a convergence module of a connection request

SYNOPSIS

```
atm_error_t xxx_connect(  
    void *bind_handle,  
    atm_addr_p addr,  
    atm_bind_handle_t myaddr,  
    atm_vc_p vc,  
    atm_uni_call_ie_p *reply,  
    atm_vc_services_p requested,  
    atm_vc_services_p *avail);
```

ARGUMENTS

<i>bind_handle</i>	Specifies the PPA binding being used. It is the value passed to the CMM when the convergence module requested a PPA bind on which the call is being processed.
<i>addr</i>	Specifies a pointer to the <code>atm_addr</code> structure, indicating the endpoint that is initiating the connection.
<i>myaddr</i>	Specifies the bind identifier for the bind point on which the call arrived. This is the value the CMM returns when the convergence module calls <code>atm_cmm_ppa_bind</code> .
<i>vc</i>	Specifies a pointer to the VC of the new connection.
<i>reply</i>	Specifies a pointer to an <code>atm_uni_call_ie</code> structure in which the convergence module writes the address of an information elements array to be used in composing the reply to the calling party. If no reply is necessary, the convergence module can ignore this argument.
<i>requested</i>	Specifies a pointer to an <code>atm_vc_services</code> structure that contains information to pass to the

`xxx_connect`

convergence module about the network resources required to establish the new connection.

avail

Specifies a pointer to an `atm_vc_services` structure that contains information about the network resources available to be committed to the new connection.

If the *avail* argument is NULL, the convergence module cannot negotiate service structure parameters; it can only accept or reject the call. If the *avail* argument is not NULL, the convergence module must evaluate the structure's contents to obtain more information about the incoming call.

For non-CBR circuits, the *avail* argument is the same as the *requested* argument. The convergence module can change negotiable parameters by modifying the appropriate fields in the `atm_vc_services` structure referenced by *avail*. In UNI 3.1, no circuit parameters can be modified. Forward and backward MTU sizes may be negotiable through the `atm_uni_call_ie` structure.

For CBR circuits, the value of *avail* depends on whether or not the CMM was able to allocate the bandwidth resources necessary to establish a new CBR connection.

If *avail* is NULL, the CMM was not able to allocate bandwidth resources. The convergence module can use the information provided in the *requested* argument to determine the resources needed to establish the connection, and can apply reserved resources to the call. You do this by setting *avail* to the address of an `atm_vc_services_t` structure that has been backed by the appropriate resources. The supplied reserved resource bit rates need not match the requested resource bit rates exactly. The forward (local system transmit) rate can be less than or equal to the requested rate (so as not to exceed the contract already established with

`xxx_connect`

the network). The backward (local system receive) rate can be greater than or equal to the requested rate (to ensure that the local system can receive at least as much as the sender contracted to send).

The convergence module can impose exact bit-rate matching, if necessary. The convergence module can change negotiable parameters by changing *avail*, as for non-CBR circuits. If the supplied reserved resource cannot be applied to the call, the convergence module receives an `ATM_CME_EP_DEAD` exception notification; the VC cause information indicates the reason for the failure.

For CBR circuits, if the *avail* argument is not NULL, the CMM was able to allocate the bandwidth resources necessary to establish a CBR connection. The convergence module can change negotiable parameters, as with non-CBR circuits. The convergence module also has the option of replacing the CMM-allocated resources with reserved resources held by the convergence module by setting *avail* to the address of the reserved resources services structure. Restrictions on bit-rate matching still apply.

DESCRIPTION

The `xxx_connect` routine is a function declared within a convergence module that the CMM calls to notify the convergence module of a connection request. The CMM calls the convergence module that is bound to the selector specified in the called party address.

The name of the routine (`xxx_connect`) can be any valid C language function name. The routine is passed to the CMM by reference only. Only one convergence module can accept a connection. The called convergence module then examines all the connection data (the information in the `atm_addr` structure as well as any IEs or other signaling information passed in) to determine if it is willing to accept the call.

If the convergence module accepts the call, the CMM and signaling module proceed with call setup. The VC is not active at this point. The connection and VC is owned by the accepting convergence module until the connection

`xxx_connect`

is destroyed; the VC's ownership cannot be changed. The convergence module VC is notified of all incoming data on the VC and exceptions on the VC. Also, the convergence module is the only module that can transmit data on the VC.

You use this routine for both point-to-point and point-to-multipoint connections.

Connections are not shared between convergence modules at the bind point. If convergence modules must share connections, one module must own the connection and coordinate access to the connection with another module. Alternatively, modules could be layered on top of a multiplexor that assumes ownership of the connection.

If the convergence module rejects the call, the CMM and signaling module release the VC and notify the calling party that the call was rejected. The CMM never accepts a call without the explicit consent of a convergence module.

NOTES

PVCs created by the `atmconfig` utility must have a convergence module specified since it is not possible for convergence modules to determine if they should accept a PVC call.

When the `atmconfig` utility creates a new PVC, the convergence module to which it belongs receives the connection notification. For PVCs, the `myaddr` argument points to the PVC PPA, the `vc` argument references the new `atm_vc` structure, and the `requested` argument references the new PVCs circuit parameters. The `addr`, `reply`, and `avail` arguments are NULL. Convergence modules cannot change or reject PVC circuit parameters; the system does not create the PVC unless it can provide the requested level of service as specified by the creator of the PVC.

When a convergence module accepts a connection, it can assume that the connection exists and that it will receive notifications if the connection is destroyed. It is free to use its private structure members in the `atm_vc` structure referenced in the `atm_addr` structure.

When a connection is accepted, it is not yet ready to carry data. The convergence module must not attempt to transmit data until it is notified that the connection is active. However, it should be ready to receive data since data can arrive ahead of the activation notification.

`xxx_connect`

EXAMPLE

See Section B.3 for a code example of the `xxx_connect` routine. In the example, `new_connect` is the name of the routine.

RETURN VALUES

If the convergence module accepts the connection, it should return `ATM_CAUSE_GOOD`. If the convergence module cannot immediately accept the connection (for example, if negotiation is needed with upper-level protocols or a user-level process), it must not block; it should return `ATM_CAUSE_DEFER`. Returning `ATM_CAUSE_DEFER` causes no immediate action to be taken on the connection since the ATM protocols allow some time for the called party to reply. When the convergence module makes a determination about the disposition of the call, it must communicate this to the CMM by using either the `atm_cmm_accept` or `atm_cmm_reject` calls. If the convergence module rejects the call, it must return an ATM error number other than `ATM_CAUSE_GOOD` or `ATM_CAUSE_DEFER`.

RELATED INFORMATION

`atm_cmm_accept`, `atm_cmm_ppa_bind`, `atm_cmm_reject`

`xxx_drop`

NAME

`xxx_drop` – Requests a signaling module to drop an endpoint in a point-to-multipoint connection

SYNOPSIS

```
atm_error_t xxx_drop(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure for the endpoint to be dropped.

DESCRIPTION

The `xxx_drop` routine is a function declared within a signaling module that the CMM calls when it needs to drop an endpoint on a point-to-multipoint connection. The name of the routine (`xxx_drop`) can be any valid C language function name. The routine is passed to the CMM by reference only.

The signaling module can obtain the information about the root's VC by following the `addr->vc` pointer. This function call is nonblocking. The CMM is notified when the connection has been dropped.

RETURN VALUES

The `xxx_drop` routine must return `ATM_CAUSE_GOOD` if the signaling module is proceeding with the endpoint drop or an ATM error number if the call cannot proceed.

RELATED INFORMATION

`atm_cmm_register_sig`

xxx_enquery

NAME

`xxx_enquery` – Requests a signaling module to obtain status for an endpoint

SYNOPSIS

```
atm_error_t xxx_enquery(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies a pointer to the `atm_addr` structure for the endpoint to be queried.

DESCRIPTION

The `xxx_enquery` routine is a function declared within a signaling module that the CMM calls when it needs status for an endpoint. The name of the routine (`xxx_enquery`) can be any valid C language function name. You pass the routine to the CMM by reference only.

This routine is nonblocking. When the query response arrives, the signaling module uses a function call to notify the CMM.

RETURN VALUES

The `xxx_enquery` routine must return `ATM_CAUSE_GOOD` if the signaling module is proceeding with the enquiry or an ATM error number if the call cannot proceed.

RELATED INFORMATION

`atm_cmm_register_sig`

`xxx_except`

NAME

`xxx_except` – Reports exceptions, errors, and system configuration changes to convergence and signaling modules

SYNOPSIS

```
int xxx_except(  
    void *handle,  
    unsigned int exception,  
    void *arg);
```

ARGUMENTS

handle Specifies an identifier that the module passed to the CMM when the module registered. For convergence and signaling modules, this is the internal handle the module registers with the CMM. This value is meaningful only to the called module.

exception Specifies a value that identifies the type of exception that occurred. The following table contains a list of valid values:

Exception	Meaning
ATM_CME_BIND_DEL	Informs the convergence module that one of its bindings is being destroyed. The usual reason for this is that a PPA is being destroyed either because its address is no longer registered with the network or because the interface to the network has gone down. The argument to this exception is the value returned from the <code>atm_cmm_ppa_bind</code> call (<code>atm_bind_handle_t</code>); the convergence module uses this value to reference the bind point. The return value from this call is ignored.
ATM_CME_CALL_FAILED	Indicates that a call to an endpoint has failed. The argument is an <code>atm_addr_p</code> that specifies the endpoint to which the CMM was trying to connect. The cause information in the <code>atm_addr</code> structure is set to indicate the reason for the call failure, which can be the failure of both a point-to-point call and the failure of adding a leaf to a point-to-multipoint call. The convergence module must destroy all references to the endpoint when processing this notification. The convergence module receives one additional notification to destroy all its references to the endpoint's VC if this was the only endpoint associated with a VC.

xxx_except

Exception	Meaning
ATM_CME_DELPVC	Informs the convergence module that a PVC has been deleted from the system. Convergence modules receive one of these exceptions for each PVC that is deleted, even PVCs that they delete. The argument for this command is an <code>atm_vc_p</code> to the PVC being deleted. When this notification is received, convergence modules must destroy all references to the PVC and must not send any more data on it.
ATM_CME_DVR_DOWN	Informs the convergence module that a driver is down, and any unused reserved resource or outstanding reservation request is revoked. The convergence module should remove knowledge of the reserved resource or outstanding reservation request services structure indicated by the <code>atm_vc_services_p</code> argument. The convergence module should not free the <code>atm_vc_services</code> structure; the CMM does this upon return from the exception notification. If the convergence module attempts to free the structure, <code>ATM_CAUSE_BUSY</code> is returned.
ATM_CME_ENQUERY_DONE	Informs the convergence module that a requested connection state enquiry function on an endpoint has completed. The argument is an <code>atm_addr_p</code> for the endpoint that was queried. The state information in the <code>atm_addr</code> structure indicates if the enquiry succeeded and the current endpoint state.
ATM_CME_EP_ACTIVE	<p>Informs the convergence module that an endpoint is now active and available for transferring data. This notification is usually the result of receiving a connection complete from the signaling module. When the convergence module requests a new connection or receives notification of a new connection, the connection is not yet ready to carry data. Only after the connection setup is complete (requiring several message exchanges between connection endpoints) will it be able to carry data.</p> <p>This command tells the convergence module that the connection is complete and ready to carry data. Use this notification for both point-to-point and point-to-multipoint notifications. The command argument is an <code>atm_addr_p</code> to the <code>atm_addr</code> structure of the endpoint that is now active.</p>

xxx_except

Exception	Meaning
ATM_CME_EP_DEAD	<p>Informs the convergence module that the connection to an endpoint has been destroyed and that all resources allocated for the endpoint should be freed. This is the only notification the convergence module will receive. After receiving this notification, it is illegal for the convergence module to reference the endpoint.</p> <p>The command argument is an <code>atm_addr_p</code> to the <code>atm_addr</code> structure of the endpoint that is down. Use this notification for both point-to-point and point-to-multipoint connections. The convergence module receives exactly one of these notifications for each endpoint associated with a VC (that is, one notification for point-to-point VCs and one notification for each endpoint of a point-to-multipoint connection).</p>
ATM_CME_ESI_DEL	<p>Informs the convergence module that an ESI it had created has been removed from the system. The CMM notifies the convergence module separately to destroy any PPAs created from the ESI. The argument is the value (<i>arg</i>) that the convergence module passed in to the <code>atm_cmm_new_esi</code> call. When this notification is received, convergence modules should destroy all references to the deleted ESI.</p>
ATM_CME_PPA_ADD	<p>Informs the convergence module that a new PPA has been configured and is available for connections. The argument is a pointer to the <code>atm_ppa</code> structure for the new PPA. If the convergence module needs to receive calls on this new PPA it must perform an <code>atm_cmm_ppa_bind</code> function call for the new PPA.</p>
ATM_CME_PPA_DEL	<p>Informs the convergence module that a PPA has been removed from the system. The CMM notifies the convergence module separately to destroy all VCs associated with the PPA. The argument for this command is a pointer to the <code>atm_ppa</code> structure for the PPA that is being deleted. When this notification is received, convergence modules should destroy all references to the deleted PPA.</p>
ATM_CME_RESV_AVAIL	<p>Informs the convergence module that an outstanding request for a reserved resource has been satisfied. The <code>atm_vc_services</code> structure indicated by the <code>atm_vc_services_p</code> argument is now a services structure that is backed by resources, or a reserved resource. The convergence module must apply this services structure to an outgoing or incoming call within the system-specified amount of time to avoid having the resources revoked by an <code>ATM_CME_RESV_EXPIRE</code> exception notification.</p>

xxx_except

Exception	Meaning
ATM_CME_RESV_EXPIRE	Notifies the convergence module that a reserved resource has not been used in the system-specified amount of time and is revoked. The convergence module should remove knowledge of the reserved resource services structure indicated by the <code>atm_vc_services_p</code> argument. The convergence module should not free the <code>atm_vc_services</code> structure; the CMM does this upon return from the exception notification. If the convergence module attempts to free the structure, <code>ATM_CAUSE_BUSY</code> is returned.
ATM_CME_RESVREQ_REL	Notifies the convergence module that an outstanding request for a reserved resource has been canceled. This exception is generated by a change in configuration parameters that causes the contents of the <code>atm_vc_services</code> structure (which were valid at the time of the request) to no longer be valid. For example, this event could occur if the system administrator reduced the per-VC bandwidth limit to values below the rates specified in the reservation request. The convergence module should remove knowledge of the reservation request services structure indicated by the <code>atm_vc_services_p</code> argument. The convergence module should not free the services structure; the CMM does this upon return from the exception notification. If the convergence module attempts to free the structure, <code>ATM_CAUSE_BUSY</code> is returned.
ATM_CME_START_VC	Notifies the convergence module that a VC on which flow was previously controlled may now resume transmission of data. The argument to this command is a pointer to the <code>atm_vc</code> structure of the VC that can resume data transmission.
ATM_CME_VC_OLD	Notifies a convergence module that a VC has aged (been inactive for a long period of time) and that the CMM is about to delete it. In this situation, the convergence module is given the chance to stop the CMM from destroying the VC. When the VC is about to be destroyed because of inactivity, this notification is made to the convergence module. The argument is an <code>atm_vc_p</code> for the VC that has aged. The convergence module may return either <code>ATM_AGER_OK</code> to allow the CMM to delete the VC or <code>ATM_AGER_RESET</code> to restart the aging timer and not to delete the VC.
ATM_SIGE_ACCEPT	Specifies that an incoming call whose disposition was previously deferred has been accepted. The signaling modules should take appropriate action to accept the call and activate the connection. The argument to this exception is an <code>atm_addr_p</code> that references the call that is being accepted.

xxx_except

Exception	Meaning
ATM_SIGE_DEL_ESI	Notifies the signaling module of a user- or convergence module-initiated deletion of an ESI (or similar) part of an address. The signaling module then performs the deletion of every PPA associated with the ESI through protocol exchanges with the switch. As each PPA is deleted, the signaling module uses the <code>atm_cmm_del_ppa</code> function call to notify the CMM. The arguments and returns are the same as for the <code>ATM_SIGE_NEW_ESI</code> exception.
ATM_SIGE_DEL_PPA	Notifies the signaling module of a user-initiated deletion of a PPA. The signaling module then performs the deletion of the PPA through protocol exchanges with the switch. The argument for this exception is a pointer to the <code>atm_ppa</code> structure of the PPA being deleted. If the signaling module returns <code>ATM_CAUSE_GOOD</code> , the CMM considers the PPA deleted (even though the protocol exchange with the switch may not be complete). It then initiates the deletion of all VCs and bind points associated with the PPA.
ATM_SIGE_NEW_ESI	Notifies the signaling module that a user- or convergence module-initiated creation of a new ESI (or similar address part) has been configured on the system. The argument for this exception is a pointer to an <code>atm_esi</code> structure for the new ESI. When a signaling module receives this exception, it should initiate creation of new PPAs based on the new ESI, notifying the CMM of each new PPA created. This exception should return <code>ATM_CAUSE_GOOD</code> if the new ESI is accepted by the signaling module. The signaling module does not need to create the new PPAs immediately (and probably cannot), so the CMM will not expect to have any new PPAs on return from this call. When a new driver is brought on line, the CMM notifies the signaling module of every ESI that the driver reports from its ROM. This enables the system to automatically configure addresses based on adapter ROM values (if any). Note that even though signaling modules have access to device driver's ROM ESI addresses from the driver's <code>atm_drv_params</code> structure, do not use those values. Use only an ESI set with this exception value in the creation of new PPAs.
ATM_SIGE_REJECT	Specifies that an incoming call whose disposition was previously deferred has been rejected. The signaling modules should take appropriate action to reject the call. The argument to this exception is an <code>atm_addr_p</code> that references the call that is being rejected.

arg

Specifies an additional argument whose interpretation depends on the exception value.

`xxx_except`

DESCRIPTION

The `xxx_except` routine is a function declared within a convergence or signaling module that the CMM calls to report exceptions, errors, and system configuration changes on a VC.

The name of the routine (`xxx_except`) can be any valid C language function name. You pass the routine to the CMM by reference only.

Exception notifications have the following characteristics:

- Convergence and signaling modules can expect them at any time.
- They are delivered in an interrupt context.
- Exception processing must not block. If a convergence or signaling module needs to defer processing an exception notification, it must arrange for a kernel thread to be run at a later time and to return immediately to the CMM.

RETURN VALUES

If a command has no explicit return value specified, it always returns `ATM_CAUSE_GOOD`; otherwise, it returns one of the specified values.

RELATED INFORMATION

`atm_cmm_register_cvg`, `atm_cmm_register_sig`

Section 7.3 for information on connection aging

Chapter 10 for more information on flow control in the ATM subsystem

xxx_manage

NAME

xxx_manage – Instructs a device driver to perform some driver management functions

SYNOPSIS

```
atm_error_t xxx_manage(  
    int unit,  
    unsigned int command,  
    void *arg);
```

ARGUMENTS

unit Specifies the unit number to which the command applies.

command Specifies the type of management function to perform. The following management functions are defined:

Function	Definition
ATM_DRVMGMT_ADDVC	Instructs the driver to create a new VC. The <i>arg</i> argument is a pointer to the <code>atm_vc</code> structure for the new VC. This call cannot block. The driver allocates resources for the VC, but does not make the VC active. The driver uses the <code>queue</code> value to assign the VC to the indicated queue (the driver should make no queue assignments other than those indicated by the CMM). The value of the <code>queue</code> member is in the range of $0 \leq \text{queue} \leq \text{maxqueue}$, where <code>maxqueue</code> is the number of queues the driver told the CMM it supports. If the driver has insufficient resources to meet the QOS specified for the VC, it must return a value other than <code>ATM_CAUSE_GOOD</code> . Note that the VPI and VCI in the <code>atm_vc_services</code> structure are not necessarily valid at this time. They become valid when the VC is enabled. This call gives the driver a chance to allocate resources for a VC before the VC is set up on the network. The connection operation cannot proceed unless all the local resources needed for the new VC are allocated.
ATM_DRVMGMT_CLEARQ	Clears the QOS parameters from a driver queue. The CMM issues this command after the last VC has been detached from a queue to indicate to the driver that the queue no longer requires servicing.
ATM_DRVMGMT_DELVC	Instructs the driver to reclaim all resources associated with a VC and to destroy any references to the VC. This call cannot block. The <i>arg</i> argument is a pointer to the <code>atm_vc</code> structure for the VC to be deleted. The driver must free all local resources for the VC and set <code>vc->drv_pp1</code> and <code>vc->drvpp2</code> to 0 prior to returning.

Function	Definition
ATM_DRVMGMT_DOWN	Instructs the interface to shut down and to stop exchanging data with the switch. This call cannot block. This call need only start the driver shutdown procedure. When the shutdown is complete, the driver must notify the CMM by using the CMM driver error interface. The CMM takes care of VC deallocation; the driver can expect to be called to deallocate local resources for each open VC. Do not use the <i>arg</i> argument.
ATM_DRVMGMT_ENBVC	Instructs the driver to enable the VC for sending and receiving data. This call cannot block. The <i>arg</i> argument is a pointer to the <i>atm_vc</i> structure, which identifies the VPI and VCI values to be activated. This call is made only after a VC is set up on the network and becomes active and only after the <i>ATM_DRVMGMT_ADDVC</i> command, never before it.
ATM_DRVMGMT_FC	Enables the types of flow control that the driver uses. The following values are supported: <i>ATM_FLOW_NONE</i> (disables all flow control); <i>ATM_FLOW_STD</i> (enables ATM Forum standard flow control); and <i>ATM_FLOW_VENDOR</i> (enables vendor-specific flow control). <i>ATM_FLOW_NONE</i> is the default state for the driver. Flow control is enabled or disabled globally rather than on a per-VC basis.
ATM_DRVMGMT_MAXVCI	Specifies the maximum VCI value for any VC on the driver. The <i>arg</i> argument is the maximum VCI value ($0 \leq \text{VCI} \leq \text{maxvci}$). This value must be less than the <i>max_vci</i> member of the <i>atm_drv_params</i> structure. The CMM uses this when it detects that the connect switch supports fewer VCIs than the driver.
ATM_DRVMGMT_MAXVPI	Specifies the maximum VPI value for any VC on the driver. The <i>arg</i> argument is the maximum VPI value ($0 \leq \text{VPI} \leq \text{maxvpi}$). This value must be less than the <i>max_vpi</i> member of the <i>atm_drv_params</i> structure. The CMM uses this when it detects that the connect switch supports fewer VPIs than the driver.
ATM_DRVMGMT_QUERY	Queries the driver for its current available resources. This call cannot block. The <i>arg</i> argument is a pointer to an <i>atm_drv_params</i> structure in which to write the information. The driver does not write the <i>num_id</i> member or the <i>ids</i> array. The values written should be the maximum allowable values for each resource minus the resources currently in use. The CMM uses this call to synchronize its current driver state information with the driver's internal state. The driver's internal state is always assumed to be correct.
ATM_DRVMGMT_RAWPARAM	Controls queuing of raw cells (not used for cooked connections). This command sets parameters that the driver uses to queue incoming raw cells into one mbuf chain rather than passing each cell up to the CMM. This reduces the cell-processing overhead when receiving raw cells. The argument is a pointer to the <i>atm_queue_param</i> structure.

xxx_manage

Function	Definition
ATM_DRVMGMT_SETQ	Specifies QOS parameters of a driver's transmit queue. Only drivers that have informed the CMM that they implement multiple queues can expect this command. The CMM issues this command before any VCs are attached to a specific queue. The argument is a pointer to an <code>atm_vc_services</code> structure that holds the queue parameters. The CMM can issue this command at any time to change the parameters of a queue.
ATM_DRVMGMT_UP	Instructs the interface to initialize and come on line. This call cannot block. This call need only start a driver initialization function. When the initialization is complete and the driver is ready to create VCs, it must use the <code>atm_cmm_error</code> routine to notify the CMM.

arg Specifies any data needed to perform the function.

DESCRIPTION

The `xxx_manage` routine is a function declared within a device driver module that the CMM calls when it needs to perform some driver management task. The name of the routine (`xxx_manage`) can be any valid C language function name. You pass the routine to the CMM by reference only.

RETURN VALUES

If the `xxx_manage` routine is completed successfully, the driver returns `ATM_CAUSE_GOOD`; otherwise, the driver returns an ATM error number indicating the cause of the failure.

RELATED INFORMATION

`atm_cmm_error`, `atm_cmm_register_dd`

xxx_mib

NAME

xxx_mib – Requests MIB information from a signaling module

SYNOPSIS

```
atm_error_t xxx_mib(  
    void *sig_handle,  
    unsigned int command,  
    atm_oid_p oid,  
    unsigned int fieldlen,  
    unsigned char *fieldval,  
    void id);
```

ARGUMENTS

<i>sig_handle</i>	Specifies the signaling module's internal handle that it provides to the CMM in the registration call.
<i>command</i>	Specifies the command to get or set a field in the MIB. Valid values are ATM_MIB_GET and ATM_MIB_SET, used to get and set a field in the MIB, respectively.
<i>oid</i>	Specifies an atm_oid structure that identifies the object (and object instance) to retrieve from the MIB.
<i>fieldlen</i>	Specifies the length of the data being provided on an ATM_MIB_SET and is ignored on an ATM_MIB_GET.
<i>fieldval</i>	Specifies a pointer to the value to be set in the indicated MIB field on an ATM_MIB_SET and is NULL on an ATM_MIB_GET.
<i>id</i>	Specifies a value that the CMM uses to identify the specific MIB request. The CMM uses this when querying a MIB value from one signaling module while sending a request to another network management entity.

`xxx_mib`

DESCRIPTION

The `xxx_mib` routine is a function declared within a signaling module that the CMM calls to access fields of MIBs maintained by the signaling module. This routine is provided only when a signaling module maintains a MIB. The values of MIB fields are returned through a call to `atm_cmm_mib_response`.

The name of the routine (`xxx_mib`) can be any valid C language function name. You pass the routine to the CMM by reference only.

The `xxx_mib` function call cannot block.

RETURN VALUES

If the value is available immediately, `atm_cmm_mib_response` is called from within the signaling module's `xxx_mib` routine and `ATM_CAUSE_GOOD` is returned. If the MIB value must be queried, `ATM_CAUSE_DEFER` must be returned and `atm_cmm_mib_response` called when the query is complete and the value is available. If a MIB set operation cannot be completed immediately, the signaling module must return `ATM_CAUSE_DEFER` and call `atm_cmm_mib_response` when the access has been completed. An error is indicated by returning the appropriate ATM error number.

RELATED INFORMATION

`atm_cmm_register_sig`

Section 5.17 for a description of the `atm_oid` structure

`xxx_mmi`

NAME

`xxx_mmi` – Connects a convergence, device driver, or signaling module to the MMI

SYNOPSIS

```
int xxx_mmi(
    void *handle,
    unsigned int command,
    void *arg,
    int *retval,
    struct ucred *cred);
```

ARGUMENTS

- handle* Specifies an identifier that the module passed to the CMM when the module registered. For device drivers, this is a unit number. For convergence and signaling modules, this is the internal handle that the module provides to the CMM. This value is meaningful only to the called module.
- command* Specifies what action the MMI routine is to perform. The format of the command is the same as that of ATM `ioctl` commands as defined in the `/usr/include/sys/atm.h` file. Commands for non-CMM ATM modules are defined using the IOW (I/O write) or equivalent macros. All commands have a type of “g” and a value of between 128 and 255 inclusive.
- arg* Specifies a command-specific argument that is interpreted by the management function based on the value of `cmd`. The format of the data referenced by this argument can be anything that is valid for `ioctl` calls.
- retval* Specifies a pointer to an integer into which the MMI routine writes a return value when returning `ESUCCESS` to the CMM. This argument is optional.
- cred* Specifies the credentials or access information passed by the system to the CMM in the `ioctl` call.

`xxx_mmi`

DESCRIPTION

The `xxx_mmi` routine is a function declared within a convergence, device driver, or signaling module that the CMM calls to pass management and configuration requests from applications through the CMM and the MMI interface.

The name of the routine (`xxx_mmi`) can be any valid C language function name. You pass the routine to the CMM by reference only.

The `xxx_mmi` routine is called from within a system call context with no external locks held. It has access to the per-process data structures and user address space of the calling process. If necessary, the `xxx_mmi` routine can block as long as it does not hold simple locks while blocking.

If a module does not require any external management or configuration capabilities, it does not have to register a management function with the CMM.

This interface follows the standard DIGITAL UNIX `ioctl` call with the following exceptions:

- The device major and minor number argument is replaced by the *handle* argument.
- No user credential information is passed. Only users with root access can use the management interface.

RETURN VALUES

If the operation completes successfully, the `xxx_mmi` routine returns `ESUCCESS`; otherwise, it returns a DIGITAL UNIX error number as defined in `sys/errno.h`. In addition, for successful completions modules can use the *retval* pointer to pass the return value from the `ioctl` system call to the calling program.

RELATED INFORMATION

`atm_cmm_register_cvg`, `atm_cmm_register_dd`,
`atm_cmm_register_sig`

Chapter 8 for information on MMI and `ioctl` calls

xxx_receive

NAME

`xxx_receive` - Passes incoming data to a convergence module

SYNOPSIS

```
void xxx_receive(  
    atm_vc_t *vc,  
    struct mbuf *mbp,  
    int length,  
    struct mbuf *trailer,  
    char pti,  
    char gfc);
```

ARGUMENTS

- vc* Specifies a pointer to the `atm_vc` structure on which the data arrived. The module being called already owns the VC, so this is used as a handle for the module to identify on which (of many) VCs the data arrived. The convergence module may use its two private structure members in the `atm_vc` structure to place local information necessary for managing the connection.
- mbp* Specifies a pointer to a list of mbufs that contain the data received.
- length* Specifies a value that indicates the exact number of bytes received. For cooked packets, this is the size of the received packet or protocol data unit (PDU) plus the padding and AAL trailer. It is always a multiple of 48. For raw data, this value is a multiple of the number of cells received. It is a multiple of 53 bytes (time-stamp bytes are not included in the count).
- trailer* Specifies a pointer to the mbuf that contains the last byte of the received data. This permits the convergence module to easily and quickly locate the AAL trailer for obtaining the PDU length and other information (such as user-to-user indications). When receiving raw cells, this argument points to the mbuf that contains the last cell in the chain.
- pti* Specifies the accumulated PTI information of a received cooked packet. The availability of this information is

`xxx_receive`

dependent on the driver's ability to keep track of all the PTI fields in all the cells that compose a cooked packet. The convergence module can determine if this argument is valid by examining the `capabilities` structure member in the device driver's `atm_drv_params` structure. This argument has no meaning for raw data.

gfc

Specifies the accumulated GFC information from the cells that make up a cooked packet. The availability of this information is dependent on the driver's ability to keep track of all the GFC fields in all the cells that compose a cooked packet. The convergence module can determine if this argument is valid by examining the `capabilities` structure member in the device driver's `atm_drv_params` structure. This argument has no meaning for raw data.

DESCRIPTION

The `xxx_receive` routine is a function declared within a convergence module that the CMM calls to pass data to a convergence module. The name of the routine (`xxx_receive`) can be any valid C language function name. You pass the routine to the CMM by reference only.

When designing receive routines, convergence module writers should remember that all data is delivered to the convergence module in an interrupt context with the processor running at the `splimp` level. The convergence module must implement a queuing policy appropriate for the convergence module's protocol. In general, convergence modules for protocols that can tolerate unspecified latency should queue the incoming data and return immediately to the CMM. Convergence modules for protocols that require bounded latencies (such as video or voice protocols) might need to do some processing before queuing the data and returning to the CMM. In either case, the receive routine must not block.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_register_cvg`

xxx_receive

Section 3.5 for a description of ATM data formats

Chapter 4 and Chapter 6 for information on device drivers and convergence modules, respectively

xxx_release

NAME

`xxx_release` – Requests a signaling module to tear down a connection

SYNOPSIS

```
atm_error_t xxx_release(  
    atm_addr_p addr);
```

ARGUMENTS

addr Specifies the pointer to the `atm_addr` structure.

DESCRIPTION

The `xxx_release` routine is a function declared within a signaling module that the CMM calls when it needs a connection torn down (a hangup). The name of the routine (`xxx_release`) can be any valid C language function name. You pass the routine to the CMM by reference only.

When the CMM makes the call, the signaling protocol module should initiate a release of the connection to the specified endpoint. This routine is nonblocking, and is used only for releasing a point-to-point connection or the root of a point-to-multipoint connection (when all leaf connections have been released).

RETURN VALUES

The `xxx_release` routine must return `ATM_CAUSE_GOOD` if the circuit deletion is completed or an ATM error number if the circuit deletion fails.

RELATED INFORMATION

`atm_cmm_register_sig`

xxx_restart

NAME

`xxx_restart` – Requests a signaling module to send a restart message

SYNOPSIS

```
atm_error_t xxx_restart(  
    void *handle,  
    unsigned int class,  
    unsigned int vpi,  
    unsigned int vci);
```

ARGUMENTS

- handle* Specifies a unique identifier that the CMM uses to identify the restart request. This has meaning only to the CMM and should not be modified by the signaling module.
- class* Specifies the type of restart that is being performed. Currently, only restarting of individual VCs or all VCs in a VP are supported. The caller sets the value of this argument to `ATM_RESTART_VP` to restart all VCs on a VP or `ATM_RESTART_VC` to restart a specific VC. These are the only two values permitted for this argument.
- vpi* Specifies which VC(s) are to be restarted. If *class* is set to `ATM_RESTART_VP`, *vpi* specifies the VP on which all VCs are to be restarted; the *vci* argument is ignored.
- vci* Specifies which VC(s) are to be restarted. If *class* is set to `ATM_RESTART_VC`, the *vpi* and *vci* arguments specify which VC is to be restarted.

DESCRIPTION

The `xxx_restart` routine is a function declared within a signaling module that the CMM calls to request a RESTART message be sent. The name of the routine (`xxx_restart`) can be any valid C language function name. You pass the routine to the CMM by reference only.

`xxx_restart`

RETURN VALUES

The `xxx_restart` routine must return `ATM_CAUSE_GOOD` if the signaling module is proceeding with the restart or an ATM error number if the call cannot proceed.

RELATED INFORMATION

`atm_cmm_register_sig`, `atm_cmm_restart_ack`

xxx_setup

NAME

`xxx_setup` – Requests a signaling module to set up a new connection

SYNOPSIS

```
atm_error_t xxx_setup(  
    atm_addr_p addr,  
    unsigned long *refptr);
```

ARGUMENTS

- addr* Specifies the pointer to the `atm_addr` structure. The CMM will have initialized the `atm_addr` structure and will have set up the `atm_uni_call_ie` structure with parameters that the protocol convergence modules supplied.
- refptr* Specifies a unique call reference number for the call. If the signaling module creates a unique call reference number for the call, the module must assign this value.

DESCRIPTION

The `xxx_setup` routine is a function declared within a signaling module that the CMM calls to request the creation of a new connection (make a call). The name of the routine (`xxx_setup`) can be any valid C language function name. You pass the routine to the CMM by reference only.

When the CMM calls this routine, the signaling protocol module should initiate a call to the party specified in the `atm_addr` structure. This routine is nonblocking, and is used only to set up point-to-point connections or the first connection in a point-to-multipoint connection.

Note

The CMM uses the value returned in *refptr* to release a connection. The signaling module should set this up as soon as possible, and before a call to `atm_cmm_con_release`.

`xxx_setup`

RETURN VALUES

The `xxx_setup` routine returns `ATM_CAUSE_GOOD` if the setup is proceeding or an ATM error number if the call cannot proceed.

RELATED INFORMATION

`atm_cmm_register_sig`

`xxx_xmit`

NAME

`xxx_xmit` – Notifies a device driver that the CMM has data to transmit

SYNOPSIS

```
atm_error_t xxx_xmit(  
    int unit,  
    struct mbuf *data,  
    long length,  
    atm_vc_p vc,  
    unsigned char clp,  
    unsigned char gfc);
```

ARGUMENTS

- | | |
|---------------|--|
| <i>unit</i> | Specifies the unit number on which to send the data. |
| <i>data</i> | Specifies the data to be transmitted. This data is passed to the driver as a chain of mbufs. The driver determines the type of data being sent by examining <code>vc->vcs->aal</code> . |
| <i>length</i> | Specifies to the driver the total number of bytes in the AAL5 PDU or the total number of bytes being sent. The use of this information depends on the driver implementation. The CMM uses this information to gather VC and interface usage statistics so the driver does not have to keep track of the total number of bytes transmitted. |
| <i>vc</i> | Specifies the VC on which to send the data. |
| <i>clp</i> | Specifies the CLP value for AAL5 PDUs. When sending raw cells, the convergence module must place this value in each cell header along with the <i>gfc</i> argument. |
| <i>gfc</i> | Specifies the GFC bits for congestion control when sending AAL5 PDUs. When sending raw cells, the convergence module must place this value in each cell header along with the <i>clp</i> argument. |

`xxx_xmit`

DESCRIPTION

The `xxx_xmit` routine is a function declared within a device driver module that the CMM calls when it has data to transmit on the interface. The name of the routine (`xxx_xmit`) can be any valid C language function name. You pass the routine to the CMM by reference only.

The `xxx_xmit` routine cannot block. The driver must either queue the data or, if the routine queue is full, return an error indication. If the data is queued, the driver must return `ATM_CAUSE_GOOD`.

If the data is not queued, the driver must return an ATM error number that indicates the reason the data was not queued (`ATM_CAUSE_QFULL` indicates a queue full condition). In addition, the driver must not discard the data. The CMM returns the error indication to the convergence module. It is up to the convergence module to implement a discard or retry policy.

RETURN VALUES

None

RELATED INFORMATION

`atm_cmm_register_dd`

B

Connection Programming Examples

This appendix contains programming code fragments for the following connection-related tasks:

- Making a call
- Adding more parties to a point-to-multipoint connection
- Processing an incoming call

B.1 Making a Call

Example B-1 shows one way to make a call for point-to-point connections and to make the first call for a point-to-multipoint connection.

Example B-1: Making a Call Code Fragment

```
make_call(unsigned char *called_party)
{
    register atm_uni_call_ie_p iep, ie;
    register union atm_cmi_addr ra;
    register atm_vc_services_p vcs;
    atm_error_t retval;
    int rv = ESUCCESS;
    extern atm_cvg_handle_t my_handle;
    extern atm_bind_handle_t my_bind;

    /* Get memory for setup IEs */
    iep = ie = atm_cmm_alloc_ie(3);
    if(ie == NULL)
    {
        rv = ENOMEM;
        goto bad;
    }

    /* Get memory for services structure */
    vcs = atm_cmm_alloc_services();
    if(vcs == NULL)
    {
        atm_cmm_free_ie(iep);
        rv = ENOMEM;
        goto bad;
    }
}
```

Example B-1: Making a Call Code Fragment (cont.)

```
/* Get the storage for the endpoint address */
ra.addr = atm_cmm_alloc_addr();
if(ra.addr == NULL)
{
    atm_cmm_free_ie(iep);
    atm_cmm_free_services(vcs);
    rv = ENOMEM;
    goto bad;
}

/* Set cell rate information for ABR traffic.
 * Cell rates - cells/sec. 1 cell = 53 bytes.
 */
vcs->fpeakcr[ATM_CLP_1] = 0x800;
vcs->bpeakcr[ATM_CLP_1] = 0x800;

vcs->valid_rates = ATM_VCRV_FPEAK1 |
ATM_VCRV_BPEAK1;

/* Best effort service class and enable tagging */
vcs->flags = ATM_SERVICES_BEI;

/* Set the Qos */
vcs->fqos = ATM_QOS_CLASSA;
vcs->bqos = ATM_QOS_CLASSA;

/* Set the MTU in the services structure */
vcs->fmtu = 1500;
vcs->bmtu = 1500;

/* Set the BB bearer class in the services structure */
vcs->bearer_class = ATM_BBEARER_BCOB_X;

/* Fill out the AAL 5 IE */
ie->ie_type = ATM_IET_AAL5;
ATM_IE_SETVAL(ie->ie.aal_params.aal5.fsdu,1500);
ATM_IE_SETVAL(ie->ie.aal_params.aal5.bsdu,1500);
ATM_IE_SETVAL(ie->ie.aal_params.aal5.mode,
ATM_AAL_MESG_MODE);
ATM_IE_SETVAL(ie->ie.aal_params.aal5.sscs,
ATM_AAL_SSCS_NULL);

/* Fill out the BLLI IE */
ie++;
ie->last = 1;
ie->ie_type = ATM_IET_BBLOW;
```

Example B-1: Making a Call Code Fragment (cont.)

```
    ATM_IE_SETVAL(ie->ie.bb_low_layer.layer2proto,
ATM_BLLI_UII2_LAN_LLC_8022);

    bcopy(called_party,ra.addr->address,ATMADDR_LEN);
    ra.addr->ton = 0;
    ra.addr->anpi = 2;

    if((retval = atm_cmm_connect(my_handle,ATM_CT_PTP, 1)
my_bind,ra,iep,ATM_AGE_FOREVER,vcs) !=
ATM_CAUSE_GOOD)
    {
        atm_cmm_free_ie(iep);
        atm_cmm_free_services(vcs);
        atm_cmm_free_addr(ra.addr);
        ATM_FREE(ivec);
        rv = EIO;
        goto bad;
    }

    /* The VC is valid if the call is proceeding! */
    ra.addr->vc->conv_pp1 =
    ra.addr->conv_p1 =
    ra.addr->conv_p2 =

    return rv;
}
```

-
- 1** For point-to-multipoint connections, use the ATM_CT_PTM argument in place of the ATM_CT_PTP argument.

B.2 Adding More Parties to a Point-to-Multipoint Connection

Example B-2 shows one way to add more parties to the VC associated with the first call in a point-to-multipoint connection.

Example B-2: Adding Parties to a Point-to-Multipoint Connection Code Fragment

```
add_leaf(atm_vc_p vc, unsigned char *new_leaf)
{
    register atm_uni_call_ie_p iep, ie;
    register struct atm_addr_p leaf;
```

Example B-2: Adding Parties to a Point-to-Multipoint Connection Code Fragment (cont.)

```
atm_error_t retval;
int rv = ESUCCESS;
extern atm_cvg_handle_t my_handle;
extern atm_bind_handle_t my_bind;

/* Get memory for setup IEs */
iep = ie = atm_cmm_alloc_ie(3);
if(ie == NULL)
{
    rv = ENOMEM;
    goto bad;
}

/* Get the storage for the endpoint address */
leaf = atm_cmm_alloc_addr();
if(leaf == NULL)
{
    atm_cmm_free_ie(iep);
    atm_cmm_free_services(vcs);
    rv = ENOMEM;
    goto bad;
}

/* Fill out the AAL 5 IE */
ie->ie_type = ATM_IET_AAL5;
ATM_IE_SETVAL(ie->ie.aal_params.aal5.fsdu,1500);
ATM_IE_SETVAL(ie->ie.aal_params.aal5.bsdu,1500);
ATM_IE_SETVAL(ie->ie.aal_params.aal5.mode,
ATM_AAL_MESG_MODE);
ATM_IE_SETVAL(ie->ie.aal_params.aal5.sscs,
ATM_AAL_SSCS_NULL);

/* Fill out the BLLI IE */
ie++;
ie->last = 1;
ie->ie_type = ATM_IET_BBLow;
ATM_IE_SETVAL(ie->ie.bb_low_layer.layer2proto,
ATM_BLLI_UTIL2_LAN_LLC_8022);

bcopy(new_leaf,leaf->address,ATMADDR_LEN);
leaf->ton = 0;
leaf->anpi = 2;

if((retval = atm_cmm_add(my_handle,leaf,ie,vc)) !=
```


Example B-2: Adding Parties to a Point-to-Multipoint Connection Code Fragment (cont.)

```
ATM_CAUSE_GOOD)
{
    atm_cmm_free_ie(iep);
    atm_cmm_free_addr(leaf);
    ATM_FREE(ivc);
    rv = EIO;
    goto bad;
}

/* The VC is valid if the call is proceeding! */
ra.addr->conv_p1 =
ra.addr->conv_p2 =

return rv;
}
```

B.3 Processing an Incoming Call

Example B-3 shows how a convergence module processes an incoming call.

Example B-3: Incoming Call Processing Code Fragment

```
atm_error_t
new_connect(void *bind_handle,
            atm_addr_p      addr,
            atm_bind_handle_t bind,
            atm_vc_p        vc,
            atm_uni_call_ie_p *reply,
            atm_vc_services_p requested,
            atm_vc_services_p *avail)
{
    register atm_uni_call_ie_p iep = NULL;
    atm_uni_call_ie_p rblli = NULL, raal = NULL;
    atm_uni_call_ie_p nblli;

    /* Verify IEs to make sure they are valid for our service */
    if(addr != NULL && (iep = addr->setup))
    {
        for(;; iep++)
        {
            switch(iep->ie_type)
            {
                /* Things we do not want to see */
                case ATM_IET_AAL1:
                case ATM_IET_AAL2:
                case ATM_IET_AAL3:
                case ATM_IET_AALU:
                    return ATM_CAUSE_APNS;

                /* Do not need to look at these */
                case ATM_IET_BBBC:
```

Example B-3: Incoming Call Processing Code Fragment (cont.)

```
case ATM_IET_BBHI:
    break;

case ATM_IET_REPEAT:
    break;

case ATM_IET_AAL5:
    /* Make sure the MTU is supported.  Accept
     * an MTU greater than 1500 and then
     * negotiate it down until larger MTUs are
     * supported.
     */
    if(ATM_IE_ISVALID(iep->ie.aal_params.aal5.fsdu))
    {
        if(ATM_IE_GETVAL(iep->ie.aal_params.aal5.fsdu)
< 1500)
            return ATM_CAUSE_APNS;
    }
    if(ATM_IE_ISVALID(iep->ie.aal_params.aal5.bsdu))
    {
        if(ATM_IE_GETVAL(iep->ie.aal_params.aal5.bsdu)
< 1500)
            return ATM_CAUSE_APNS;
    }
    raal = iep;
    break;

case ATM_IET_BBLow:
    /* Check for LLC encapsulation */
    if(ATM_IE_ISVALID(iep->ie.bb_low_layer.layer2proto))
    {
        if(ATM_IE_GETVAL(iep->ie.bb_low_layer.layer2proto)
!= 0x0C)
            return ATM_CAUSE_BCNI;
        rblli = iep;
        break;
    }
    break;
}
if(iep->last)
    break;
}

/* Make sure the required info is present */
if(rblli == NULL) /* No required BLLI */
{
    addr->diag_length = 2;
    addr->diagnostic[0] = 0x5;
    addr->diagnostic[1] = 0x5e;
    return ATM_CAUSE_CR;
}
if(raal == NULL) /* No required AAL parameters */
{
    addr->diag_length = 2;
    addr->diagnostic[0] = 0x5;
    addr->diagnostic[1] = 0x58;
    return ATM_CAUSE_CR;
}

/* Set up reply values */
```

Example B-3: Incoming Call Processing Code Fragment (cont.)

```
    iep = atm_cmm_alloc_ie(2);
    if(!iep)
        return ATM_CAUSE_RUU;
    bcopy(raal, iep, sizeof(atm_uni_call_ie_t));
    ATM_IE_SETVAL(iep->ie.aal_params.aal5.fsdu, 1500);
    ATM_IE_SETVAL(iep->ie.aal_params.aal5.bsdu, 1500);
    nblli = iep+1;
    bcopy(rblli, nblli, sizeof(atm_uni_call_ie_t));
    nblli->last = 1;
    *reply = iep;

    /* Now take care of the services stuff */
    if ( (avail) && (*avail) ) {
        (*avail)->fmtu = (*avail)->bmtu = 1500;
    }
}

vc->conv_pp1 =
vc->conv_pp2 =

/* If there is no signalling protocol, then this is a PVC */
if(vc->ppa->sig != NULL)
{
    addr->conv_p1 =
    addr->conv_p2 =
}

return ATM_CAUSE_GOOD;
}
```

Index

A

- aal member, 3-28
- aal_params member, 3-34
- adapter
 - installation, 1-5
- addr member, 6-11
- address member
 - in the atm_addr structure, 3-20
 - in the atm_ppa structure, 3-39
- Address Resolution Protocol
 - (*See ARP*)
- addrlen member, 3-38
- AESA, 3-5
- anpi member
 - in the atm_addr structure, 3-20
 - in the atm_ppa structure, 3-38
- ARP
 - specifying server ATM address, 2-8
 - specifying server IP address, 2-8
 - specifying system role, 2-8
- ARP server
 - and /etc/atmhosts file, 2-13
- ARP table
 - and CLIP, 2-27
 - creating entries, 2-27
 - deleting entries, 2-27
 - displaying for LANE, 2-28
- Asynchronous Transfer Mode
 - (*See ATM*)
- ATM
 - adapter, 1-5, 2-1
 - architecture, 1-1
 - cells, 3-3
 - CMM routines, A-1
 - configuring the software, 2-1
 - connection management, 7-1
 - convergence module interface, 6-1
 - device driver interface, 4-1
 - error codes, 3-2
 - flow control, 10-1
 - global data structures, 3-14
 - header files, 3-1
 - locking macros, 3-10
 - Module Management Interface, 8-1
 - programming examples, B-1
 - queuing guidelines, 9-1
 - signaling module interface, 5-1
 - subsets not installed, 2-30
 - tuning, 11-1
- ATM address
 - adding , 6-8
 - data structure, 3-18
 - deleting, 6-9
- ATM address structure
 - (*See atm_addr_structure*)
- ATM architecture
 - overview, 1-1
- ATM End System Address
 - (*See AESA*)
- ATM environment
 - management tasks, 2-26
- ATM problems
 - ATM not configured in kernel, 2-30
 - driver not configured, 2-31
 - subsets not installed, 2-30
- ATM software
 - configuring, 2-1
 - verifying kernel options, 2-5
 - verifying subsets are installed, 2-5
- ATM subsystem

- adapter installation, 1-5
- configuring, 1-5
- general features, 3-1
- interfaces, 1-4
- locking macros, 3-10
- tuning, 11-1
- types of circuits, 3-14
- atm.h file, 3-2
 - error codes, 3-2
- atm_addr structure, 3-18
 - allocating memory for, 3-22
- atm_adi.h header file, 3-1
- atm_cause_info structure, 3-40
- atm_cmi.h header file, 3-1
- atm_cmi_addr union, 6-10
- atm_cmm_accept kernel routine, 6-8
 - description, A-2
 - function definition, A-2
- atm_cmm_activate_con kernel routine, 5-2
 - description, A-4
 - function definition, A-4
- atm_cmm_add kernel routine, 6-5
 - description, A-6
 - function definition, A-6
 - use in code fragment, B-3
- atm_cmm_adi_set_cause kernel routine
 - description, A-8
 - function definition, A-8
- atm_cmm_adi_set_log kernel routine
 - description, A-10
 - function definition, A-10
- atm_cmm_alloc_addr call, 3-22
- atm_cmm_alloc_addr kernel routine
 - description, A-12
 - function definition, A-12
- atm_cmm_alloc_ie call, 3-34
- atm_cmm_alloc_ie kernel routine
 - description, A-13
 - function definition, A-13
- atm_cmm_alloc_services call, 3-31
- atm_cmm_alloc_services kernel routine
 - description, A-15
 - function definition, A-15
- atm_cmm_bind_info kernel routine, 6-3
 - description, A-16
 - function definition, A-16
- atm_cmm_con_deleted kernel routine, 5-3
 - description, A-21
 - function definition, A-21
- atm_cmm_con_failed kernel routine, 5-3
 - description, A-23
 - function definition, A-23
- atm_cmm_con_release kernel routine, 5-3
 - description, A-25
 - function definition, A-25
- atm_cmm_connect kernel routine, 6-4
 - description, A-27
 - function definition, A-27
 - use in code fragment, B-1
- atm_cmm_del_esi kernel routine, 6-9
 - description, A-31
 - function definition, A-31
- atm_cmm_del_ppa kernel routine, 5-5
 - description, A-32
 - function definition, A-32
- atm_cmm_drop kernel routine, 6-6
 - description, A-34
 - function definition, A-34
- atm_cmm_enquery kernel routine, 6-6
 - description, A-35
 - function definition, A-35
- atm_cmm_ep_dropped kernel routine, 5-3
 - description, A-36
 - function definition, A-36
- atm_cmm_error kernel routine, 4-2

- description, A-38
- function definition, A-38
- atm_cmm_find_driver kernel routine
 - description, A-42
 - function definition, A-42
- atm_cmm_findaddr kernel routine, 5-4
 - description, A-40
 - function definition, A-40
- atm_cmm_free_addr call, 3-23
- atm_cmm_free_addr kernel routine
 - description, A-43
 - function definition, A-43
- atm_cmm_free_ie call, 3-36
- atm_cmm_free_ie kernel routine
 - description, A-44
 - function definition, A-44
- atm_cmm_free_services kernel routine, 6-4
 - description, A-45
 - function definition, A-45
- atm_cmm_mib_response kernel routine, 5-5
 - description, A-47
 - function definition, A-47
- atm_cmm_new_call kernel routine, 5-2
 - description, A-49
 - function definition, A-49
 - use in accepting connections, 7-5
- atm_cmm_new_esi kernel routine, 6-8
 - description, A-52
 - function definition, A-52
- atm_cmm_new_ppa kernel routine, 5-4
 - description, A-54
 - function definition, A-54
- atm_cmm_new_thread kernel routine
 - description, A-57
 - function definition, A-57
- atm_cmm_next_cause kernel routine
 - description, A-59
 - function definition, A-59
- atm_cmm_oam_receive kernel routine, 4-2
 - description, A-61
 - function definition, A-61
- atm_cmm_ppa_bind kernel routine, 6-6
 - description, A-62
 - function definition, A-62
- atm_cmm_ppa_info kernel routine, 6-3
 - description, A-65
 - function definition, A-65
- atm_cmm_ppa_unbind kernel routine, 6-8
 - description, A-69
 - function definition, A-69
- atm_cmm_receive kernel routine, 4-2
 - description, A-70
 - function definition, A-70
- atm_cmm_register_cvg kernel routine, 6-1
 - description, A-73
 - function definition, A-73
- atm_cmm_register_dd kernel routine, 4-1
 - description, A-76
 - function definition, A-76
- atm_cmm_register_sig kernel routine, 5-2
 - description, A-78
 - function definition, A-78
- atm_cmm_reject kernel routine, 6-8
 - description, A-82
 - function definition, A-82
- atm_cmm_release kernel routine, 6-5
 - description, A-83
 - function definition, A-83
- atm_cmm_reply kernel routine, 5-2
 - description, A-85
 - function definition, A-85

atm_cmm_reserve_resources
 kernel routine, 6-4
 description, A-87
 function definition, A-87
 use in making connections, 7-2
 atm_cmm_restart kernel routine,
 5-3
 description, A-89
 function definition, A-89
 atm_cmm_restart_ack kernel
 routine, 5-4
 description, A-91
 function definition, A-91
 atm_cmm_send kernel routine, 6-6
 description, A-93
 function definition, A-93
 atm_cmm_set_cause kernel routine
 description, A-95
 function definition, A-95
 atm_cmm_set_log kernel routine
 description, A-97
 function definition, A-97
 atm_cmm_smi_set_cause kernel
 routine
 description, A-99
 function definition, A-99
 atm_cmm_smi_set_log kernel
 routine
 description, A-101
 function definition, A-101
 atm_cmm_status_done kernel
 routine, 5-4
 description, A-103
 function definition, A-103
 atm_cmm_unregister_cvg kernel
 routine, 6-3
 description, A-104
 function definition, A-104
 atm_cmm_vc_control kernel
 routine, 6-6
 description, A-106
 function definition, A-106
 atm_cmm_vc_get kernel routine,
 5-5
 description, A-108
 function definition, A-108
 atm_cmm_vc_stats kernel routine,
 6-9
 description, A-109
 function definition, A-109
 atm_data.c file, 3-2
 atm_drv_params structure, 4-3
 atm_error member, 3-21
 atm_esi structure, 3-39
 ATM_FREE macro, 3-9
 ATM_MALLOC macro, 3-9
 atm_mmi_path structure, 8-3
 atm_oid structure, 5-5
 atm_osf.h header file, 3-1
 atm_ppa structure, 3-37
 atm_queue_param structure, 4-7
 ATM_REVISION constant, 3-2
 atm_smi.h header file, 3-1
 atm_uni_call_ie structure, 3-31
 accessing an element in the
 array, 3-34
 allocating memory for, 3-34
 atm_vc structure, 3-15
 atm_vc_services structure, 3-23
 allocating memory for, 3-31
 atm_vc_stats structure, 6-9
 atmarp command, 2-27
 atmconfig command, 2-27
 and PVCs, 3-7
 starting the ATM subsystem,
 2-18, 2-26
 atmelan command, 2-28
 atmifmp command, 2-28
 atmsetup script
 and IP switching, 2-23
 configuring CLIP, 2-15
 configuring LANE, 2-21
 atmsig command, 2-27
 attach routine, 4-1
 attribute
 (*See kernel attribute*)

B

bb_high_layer member, 3-34

- bb_low_layer member, 3-34
- bbbc member, 3-34
- bburstcr member, 3-26
- bearer_class member, 3-28
- bind
 - information types, A-16
- bmtu member, 3-25
- bpeakcr member, 3-26
- bqos member, 3-25
- bsustcr member, 3-26
- bsysresvlim kernel attribute, 11-1
- bytes_in member, 6-10
- bytes_out member, 6-10

C

- call
 - (*See incoming call*)
- call_reference member, 3-17
- capabilities member, 4-6
- cause member, 3-22, 3-41
- cell
 - defined, 3-3
 - loss, 3-5
 - OAM, 4-2
 - raw, 3-3
 - receiving, 4-2
 - time-stamping of, 3-4
- circuit
 - types of, 3-14
- Classical IP
 - (*See CLIP*)
- CLIP
 - configuration steps, 2-13
 - configuring LIS interfaces, 2-16
 - description, 2-1
 - editing /etc/atm.conf file, 2-16
 - editing /etc/atmhosts file, 2-13
 - editing /etc/hosts file, 2-14
 - planning for configuration, 2-6
 - running atmconfig, 2-18
 - running atmsetup, 2-15
 - setting up PVCs, 2-16
- CLIP problems

- connection terminates
 - abnormally, 2-34
 - host is unreachable, 2-34
 - lis interfaces not configured, 2-32
 - lis interfaces not created, 2-31
 - remote host is unknown, 2-33
 - signaling not configured, 2-31
- CMM
 - interfaces, 1-4
 - overview, 1-4
- cmm_drv_handle member, 3-30
- configuration
 - planning for CLIP, 2-6
 - planning for IP switching, 2-11
 - planning for LANE, 2-9
- connect member, 3-21
- connection
 - accepting, 7-5
 - activating, 5-2
 - controlling aging of, 7-6
 - deleting, 5-3
 - making in ATM, 7-1
 - outgoing, 7-1
 - point-to-multipoint, 7-1
 - point-to-point, 7-1
 - receiving notification of, 6-7
 - releasing, 5-3, 7-8
 - reporting a failure, 5-3
 - requesting endpoint
 - information, 6-6
 - reserving resources, 7-2
 - tearing down, 6-5
 - terminates abnormally (CLIP), 2-34
 - terminates abnormally (IP switching), 2-41
 - terminates abnormally (LANE), 2-37
 - type of, A-27
- Connection Management Module
 - (*See CMM*)
- conv_p1 member, 3-22
- conv_p2 member, 3-22
- conv_pp1 member, 3-17
- conv_pp2 member, 3-17

- converge_handle member, 3-31
- convergence module, 1-3
 - flow control, 10-3
 - MMI, 8-6
 - queuing guidelines, 9-3
 - receiving exception notification, 6-2
 - registering with CMM, 6-1
 - requesting interface parameters, 6-3
 - reserving resources, 6-4
 - structures, 6-9
 - unregistering with CMM, 6-3
- cooked data, 3-4

D

- data
 - carrying, 3-4
 - cooked, 3-4
 - formats of, 3-3
 - receiving on a virtual circuit, 6-2
- data packet
 - receiving, 4-2
- data structure
 - ATM, 3-14
 - global, 3-14
 - logging, 3-40
- device driver
 - and lost cells, 3-5
 - and QOS, 9-1
 - and time-stamping, 3-5
 - ATM structures, 4-2
 - disabling, 2-27
 - enabling, 2-27
 - interface, 4-1
 - management functions, A-124
 - MMI, 8-5
 - not configured, 2-31
 - queuing guidelines, 9-1
 - registering with CMM, 4-1
 - structures, 4-2
 - types of errors, A-38
- diag member, 3-42
- diag_length member, 3-22, 3-42

- diagnostic member, 3-22
- direction member, 3-18
- driver member
 - in the atm_esi structure, 3-40
 - in the atm_ppa structure, 3-38
- dropped member, 4-5
- drv_pp1 member, 3-17
- drv_pp2 member, 3-17
- drv_resource member, 3-30

E

- ELAN
 - and /etc/hosts file, 2-20
 - configuring interfaces, 2-22
 - specifying name, 2-10
 - specifying number, 2-10
- emulated LAN
 - (*See ELAN*)
- end system
 - definition, 2-1
- end system identifier
 - (*See ESI*)
- endpoint
 - creating, 6-5
 - dropping, 5-3, 6-6
 - requesting connection state information, 6-6
 - requesting information, 5-4
- endpoint member, 3-21
- endstate member, 3-22
- eprtype member, 3-20
- errno member, 3-17
- error
 - data structure, 3-40
 - reporting to CMM, 4-2
- error codes, 3-2
- errors
 - reported by device driver, A-38
- ESI
 - creating, 2-27
 - data structure, 3-39
 - definition, 3-5
 - destroying, 2-27
 - specifying for CLIP, 2-7

- specifying for LANE, 2-9
- esi member, 3-40
- esi_arg member, 3-39
- esilen member, 3-40
- /etc/atm.conf file
 - editing for CLIP, 2-16
 - editing for IP switching, 2-24
- /etc/atmhosts file
 - and ARP server, 2-13
 - editing for CLIP, 2-13
 - editing for LANE, 2-19
- /etc/hosts file
 - and ELAN hosts, 2-20
 - editing for CLIP, 2-14
 - editing for IP switching, 2-23
 - editing for LANE, 2-20
- exception
 - receiving, 6-2
 - types of, A-118

F

- fburstcr member, 3-26
- flags member, 3-27, 4-8
- flow, 2-3
 - displaying information about, 2-28
- flow control, 10-1
 - convergence module, 10-3
 - hardware, 10-1
 - software, 10-2
 - specifying for CLIP, 2-8
 - specifying for LANE, 2-10
- flowcontrol member, 4-5
- fmtu member, 3-25
- fpeakcr member, 3-26
- fqos member, 3-25
- fsustcr member, 3-26
- fysresvlim kernel attribute, 11-1

G

- global data structure, 3-14
- GUIC_MMI_PATH command, 8-5

H

- hard_mtu member, 4-5
- header file, 3-1
- host is unreachable message
 - CLIP, 2-34
 - IP switching, 2-40
 - LANE, 2-37

I

- ids member, 4-7
- IE
 - reading and writing, 3-36
- ie_type member, 3-33
- incoming call
 - accepting, 6-8
 - receiving, 5-2
 - rejecting, 6-8
- information element
 - (*See IE*)
- interface
 - configuring for IP switching, 2-24
 - not configured (CLIP), 2-32
 - not configured (IP switching), 2-38
 - not configured (LANE), 2-36
 - not created (CLIP), 2-31
 - not created (IP switching), 2-38
 - not created (LANE), 2-35
- interface number
 - specifying for IP switching, 2-12
- interface parameter
 - requesting, 6-3
- Internet Protocol
 - (*See IP switching*)
- ioctl command
 - (*See MMI*)
- IP switching
 - configuration steps, 2-22
 - description, 2-3
 - disabling, 2-28
 - displaying configuration, 2-28
 - displaying flow information, 2-28
 - displaying statistics, 2-28

- editing /etc/atm.conf file, 2-24
- editing /etc/hosts file, 2-23
- enabling, 2-28
- planning for configuration, 2-11
- running atmconfig, 2-26
- running atmsetup, 2-23
- setting up interfaces, 2-24

IP switching problems

- connection terminates
 - abnormally, 2-41
- host not reachable, 2-40
- ips interfaces not configured, 2-38
- ips interfaces not created, 2-38
- ping command fails, 2-40
- remote host is unknown, 2-39

K

kernel

- installing ATM options, 2-5
- tuning for ATM, 11-1
- verifying ATM options, 2-5

kernel attribute

- bsysresvlim, 11-1
- fsysresvlim, 11-1
- nelan, 11-2
- resvtimeo, 11-1
- sysrefvlim, 11-1
- timer_resolution, 11-2

kernel data file, 3-2

L

LAN emulation
(See *LANE*)

- configuration steps, 2-19

LANE

- configuring ELAN interfaces, 2-22
- description, 2-2
- displaying ARP table, 2-28
- editing /etc/atmhosts file, 2-19
- editing /etc/hosts file, 2-20
- management tasks, 2-28

- planning for configuration, 2-9
- running atmsetup, 2-21

LANE problems

- connection terminates
 - abnormally, 2-37
- elan interfaces not configured, 2-36
- elan interfaces not created, 2-35
- host is unreachable, 2-37
- remote host is unknown, 2-36
- signaling not configured, 2-35

last member, 3-33

last_in member, 6-10

last_out member, 6-10

learp command, 2-28

LECS

- configuring, 2-28
- creating, 2-28
- displaying counters, 2-28
- displaying parameters, 2-28
- displaying status, 2-28

lerrstat member, 3-28

LES

- and /etc/atmhosts file, 2-19
- specifying ATM address, 2-10

LIS

- configuring interfaces, 2-16
- creating, 2-27
- definition, 2-1
- specifying numbers, 2-8

location member, 3-22, 3-42

locking macros, 3-10

lost cells, 3-5

M

management functions

- device driver, A-124

Management Information Base
(See *MIB*)

management tasks, 2-26

max_vci member, 4-4

max_vcib member, 4-4

max_vpi member, 4-4

max_vpib member, 4-4

mbuf, 3-4

memory

- allocating for ATM, 3-9
- allocating for atm_addr structure, 3-22
- allocating for atm_uni_call_ie structure, 3-34
- allocating for atm_vc_services structure, 3-31
- allocation of, 3-9
- freeing, 3-23

MIB

- notifying CMM of access, 5-5

MMI

- calling conventions, 8-5
- connecting convergence module to, 6-2
- convergence module, 8-6
- creating path, 8-2
- defining ioctl commands, 8-3
- device driver, 8-5
- signaling module, 8-6

Module Management Interface
(See *MMI*)

module_name member, 3-42

MTU

- in incoming calls, 7-5
- in outgoing calls, 7-4
- specifying size, 2-11

N

name member, 4-4

nelan kernel attribute, 11-2

nerrstat member, 3-30

netsetup script

- and CLIP interfaces, 2-16
- and LANE interfaces, 2-22

network mask

- for IP switching, 2-4

nqueue member, 4-5

num_vc member, 4-4

num_vci member, 4-5

num_vpi member, 4-5

numid member, 4-7

O

OAM cell, 4-2

- passing to CMM, A-61, A-71

opened member, 6-10

P

packets_in member, 6-10

packets_out member, 6-10

permanent virtual circuit
(See *PVC*)

physical point of attachment
(See *PPA*)

ping command

- fails, 2-40

point-to-multipoint
(See *connection*)

point-to-point
(See *connection*)

PPA, 3-5

- adding, 5-4
- binding to, 3-6, 6-6
- data structure, 3-37
- deleting, 5-5
- information types, A-65
- permanent virtual circuit, 3-7
- switched virtual circuit, 3-8
- unbinding, 6-8

ppa member, 3-18

ppas_id member, 3-38

probe routine, 4-1

Problems
(See *ATM problems, CLIP problems, IP switching problems, and LANE problems,*)

PVC

- configuring for CLIP, 2-16
- creating, 2-27, 7-9, A-29
- destroying, 2-27
- PPAs, 3-7
- verifying creation, 2-19

Q

qlength member, 4–8

QOS

and convergence modules, 9–3

and device drivers, 9–1

clearing, A–124

definition, 3–25

modifying, 6–6

values, 3–25

qtime member, 4–8

quality of service

(*See QOS*)

queue member, 3–28

queuing

and convergence module, 9–3

and device driver, 9–1

R

rates member, 4–5

raw ATM cell, 3–3

reason member, 3–42

received member, 4–5

remote host

unknown (CLIP), 2–33

unknown (IP switching), 2–39

unknown (LANE), 2–36

unreachable (CLIP), 2–34

unreachable (IP switching), 2–40

unreachable (LANE), 2–37

remote system

requesting connection to, 6–4

resource

releasing, 6–4

reserving, 6–4

reserving for outgoing call, 7–2

restart

reporting to CMM, 5–4

resvtimeo kernel attribute, 11–1

route

(*See also static route*)

update methods, 2–12

S

SDH

specifying for CLIP, 2–8

specifying for LANE, 2–10

selector byte, 3–5

selector member, 3–18

sent member, 4–4

setup member, 3–21

sig member, 3–38

sig_p1 member, 3–22

sig_p2 member, 3–22

sig_pp1 member, 3–17

sig_pp2 member, 3–17

signaling

managing, 2–27

not configured (CLIP), 2–31

not configured (LANE), 2–35

specifying for CLIP, 2–8

signaling module

interface, 5–1

MMI, 8–6

registering with CMM, 5–2

structure, 5–5

sigp1 member, 3–40

sigp2 member, 3–40

SNAP

specifying default VCI, 2–12

SONET

specifying for CLIP, 2–8

specifying for LANE, 2–10

state member, 3–21

static route, 2–12

adding for CLIP, 2–18

adding for IP switching, 2–25

status enquiry

reporting to CMM, 5–4

structure

(*See also union*)

atm_addr, 3–18

atm_cause_info, 3–40

atm_drv_params, 4–3

atm_esi, 3–39

atm_mmi_path, 8–3

atm_oid, 5–5

- atm_ppa, 3-37
- atm_queue_param, 4-7
- atm_uni_call_ie, 3-31
- atm_vc, 3-15
- atm_vc_services, 3-23
- atm_vc_stats, 6-9
- subaddress member, 3-20
- subaddress_type member, 3-20
- Subnetwork Attachment Point
(*See SNAP*)
- subset
 - not installed, 2-30
 - verifying installation, 2-5
- SVC
 - destroying, 2-27
 - PPAs, 3-8
- switch
 - definition, 2-1
- Synchronous Data Hierarchy
(*See SDH*)
- Synchronous Optical Network
(*See SONET*)
- sysresvlim kernel attribute, 11-1

T

- time-stamp, 3-5
- timer_resolution kernel attribute,
11-2
- ton member
 - in the atm_addr structure, 3-20
 - in the atm_ppa structure, 3-38
- tuning, 11-1
- type member, 4-4
 - in the atm_ppa structure, 3-39

U

- UNI
 - version number for CLIP, 2-8
 - version number for LANE, 2-10
- uni member, 3-39
- union
 - atm_cmi_addr, 6-10

- unit member, 4-4
- unknown host message
 - CLIP, 2-33
 - IP switching, 2-39
 - LANE, 2-36
- User-Network Interface
(*See UNI*)

V

- valid_rates member, 3-26
- VC
 - (*See also PVC*)
 - (*See also SVC*)
 - activating, 5-2
 - controlling aging of, 6-6
 - creating permanent, 7-9
 - creating signaling, 7-10
 - data structure, 3-15
 - enabling accounting, 2-8, 2-10
 - modifying parameters, 6-6
 - modifying QOS parameters, 6-6
 - monitoring, 2-27
 - ownership of , 6-5, 6-7, A-113
 - QOS values, 3-25
 - requesting statistics for, 6-9
 - requesting status for, 5-5
 - restarting, 5-3
 - transmitting data on, 6-6
 - types of operations on, A-107
- vc member, 4-8
 - in the atm_addr structure, 3-20
 - in the atm_vc structure, 3-25
- VC status
 - reporting to CMM, 5-4
- vci member, 3-18
- vcn member, 6-11
- vcs member, 3-17
- virtual circuit
(*See VC*)
 - ownership of, A-29
- vpi member, 3-18

X

- xxx_add kernel routine
 - description, A-110
 - function definition, A-110
- xxx_connect kernel routine, 6-7
 - description, A-111
 - function definition, A-111
 - use in code fragment, B-5
- xxx_drop kernel routine
 - description, A-116
 - function definition, A-116
- xxx_enquery kernel routine
 - description, A-117
 - function definition, A-117
- xxx_except kernel routine, 6-2
 - description, A-118
 - function definition, A-118
- xxx_manage kernel routine
 - description, A-124
 - function definition, A-124
- xxx_mib kernel routine
 - description, A-127
 - function definition, A-127
- xxx_mmi kernel routine, 6-2
 - description, A-129
 - function definition, A-129
- xxx_receive kernel routine, 6-2
 - description, A-131
 - function definition, A-131
- xxx_release kernel routine
 - description, A-134
 - function definition, A-134
- xxx_restart kernel routine
 - description, A-135
 - function definition, A-135
- xxx_setup kernel routine
 - description, A-137
 - function definition, A-137
- xxx_xmit kernel routine
 - description, A-139
 - function definition, A-139

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—	Local Digital subsidiary or approved distributor
Internal (submit an Internal Software Order Form, EN-01740-07)	—	SSB Order Processing – NQO/V19 <i>or</i> U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

Reader's Comments

DIGITAL UNIX

Asynchronous Transfer Mode
AA-QDP5D-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: readers_comment@zk3.dec.com
- Fax: (603) 881-0120, Attn: UEG Publications, ZK03-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Usability (ability to access information quickly)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name, title, department _____

Mailing address _____

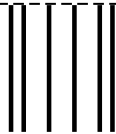
Electronic mail _____

Telephone _____

Date _____

----- Do Not Cut or Tear - Fold Here and Tape -----

digitalTM



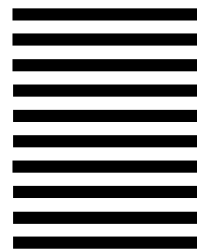
NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UBPG PUBLICATIONS MANAGER
ZKO3 3/Y32
110 SPIT BROOK RD
NASHUA NH 03062 9987



----- Do Not Cut or Tear - Fold Here -----

Cut on Dotted Line