SunATM™ Application Programmer's Interface and Man Pages

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300   Fax 650 969-9131

Part No. 805-4477-10
August 1998, Revision A

Send comments about this document to: `docfeedback@sun.com`

Contents

Figures

Tables

Code Samples

Preface

SunATM Application Programmer's Interface and Man Pages combines Appendix E, "Application Programmer's Interface," of the *SunATM 3.0 Installation and User's Guide* (805-0331-10) and the man pages that were shipped with the SunATM™ 3.0 software.

Note – This manual does not contain any hardware or software installation instructions. For these instructions, refer to the *SunATM 3.0 Installation and User's Guide*.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- *Solaris 2.x Handbook for SMCC Peripherals*
- AnswerBook™ online documentation for the Solaris™ 2.x software environment
- Other software documentation that you received with your system

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output.	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail`.

`AaBbCc123`	What you type, when contrasted with on-screen computer output.	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized. Command-line variable; replace with a real name or value.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be `root` to do this. To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Installation and Service	*SunATM 3.0 Installation and User's Guides*	801-0331
Release Information	*SunATM 3.0 Release Notes*	801-3472

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

`smcc-docs@sun.com`.

Please include the part number of your document in the subject line of your email.

CHAPTER 1

Application Programmers' Interface

The Application Programmers' Interface (API) that is provided with this software release is an interim API from Sun which can be used on Sun Platforms.

In the ATM environment, data is sent between hosts over Virtual Circuits (VCs). VCs are point-to-point (or point-to-multipoint) connections between two or more ATM hosts.

VCs can be created in one of two ways:

- Manual configuration at each host and each intermediate network point, also known as Permanent Virtual Circuits (PVC)
- ATM signalling, also known as Switched Virtual Circuits (SVC)

The ATM Forum's User Network Interface protocol is based on the ITU's Q.2931 specification.

After the VC has been determined, the application must notify the SunATM ba driver that it will be sending and receiving data on the new VC.

- If using a PVC, this is the only configuration required on the Sun host.
- If using an SVC, there are two required actions:
  - Create the SVC with the q93b driver.
  - Establish the data connection with the ba driver.

Note – For historical reasons, Q.93B and Q.2931 are used interchangeably.

Using the SunATM API with the q93b and the ATM Device Drivers

The architecture illustrated in FIGURE 1-1 must be established on a SunATM system in order to perform Q.2931 signalling and send data over established connections. The ATM device driver, SSCOP modules, and q93b driver are "plumbed" at boot time. The task remaining for application developers is to create the connections between their application and the q93b and ATM device drivers.

Both the q93b and ATM device driver are STREAMS drivers; connecting to them is for the most part no different than connecting to other STREAMS drivers. The following sections describe the steps required to connect to each driver, use the drivers to establish ATM connections, and send data over those connections.

For examples of user applications that use the SunATM API, see the sample programs installed in `/opt/SUNWatm/examples`

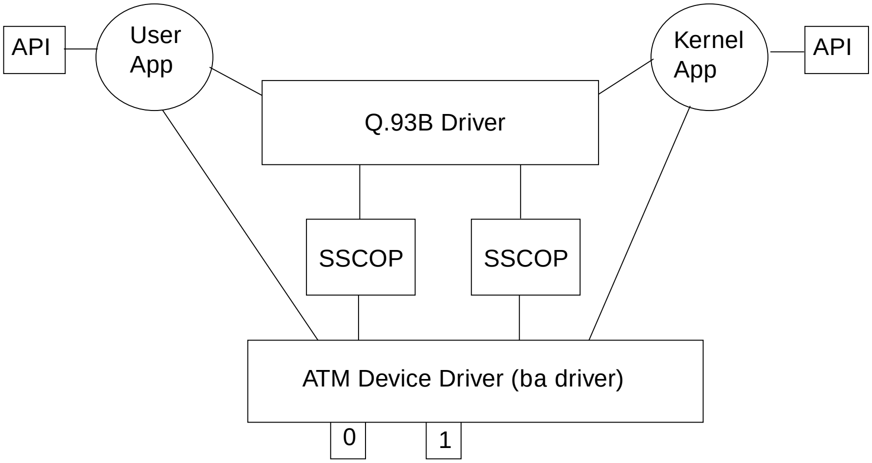

FIGURE 1-1 ATM Signalling

Q.93b Driver Interface

The signalling API, called Q.2931 Call Control (qcc), consists of two sets of similar functions: one for applications running in the kernel, and one for applications running in user space. Each set provides functions to build and parse Q.2931 signalling messages, which are required to set up and tear down connections.

One additional function is provided to assist applications in establishing appropriate connections to the q93b driver. q_ioc_bind associates a service access point (SAP) with the specified connection to the q93b driver. The SAP is used by the driver to direct incoming messages to applications.

Establishing a Connection to the q93b Driver

The open(2) system call should be used first to obtain a file descriptor to the driver. After opening the driver, q_ioc_bind should be called, associating in the q93b driver a service access point (SAP) with this application. Finally, if the application is a kernel driver, it should be linked above the q93b driver, using the I_LINK or I_PLINK ioctl (refer to the streamio(7) man page for information about this ioctl).

Setting up an ATM Connection Over a Switched Virtual Circuit (SVC)

After connecting to the q93b driver, either by directly calling the functions as a user application, or by having a setup program connect your application driver as described in the preceding section, the q93b driver is available to your application to establish switched virtual circuits (SVCs) using the Q.2931 signalling protocol. The Q.2931 message set is displayed in TABLE 1-1.

TABLE 1-1 Messages Between the User and the q93b Driver

Message Type	Direction*
SETUP	BOTH
SETUP_ACK	UP
CALL_PROCEEDING	BOTH
ALERTING	BOTH
CONNECT	BOTH
CONNECT_ACK	UP
RELEASE	DOWN
RELEASE_COMPLETE	BOTH
STATUS_ENQUIRY	DOWN
STATUS	UP

*UP is from q93b to user;
DOWN is from user to q93b

TABLE 1-1 Messages Between the User and the q93b Driver

Message Type	Direction*
NOTIFY	BOTH
RESTART	BOTH
RESTART_ACK	BOTH
ADD_PARTY	BOTH
ADD_PARTY_ACK	BOTH
ADD_PARTY_REJECT	BOTH
PARTY_ALERTING	BOTH
DROP_PARTY	BOTH
DROP_PARTY_ACK	BOTH
LEAF_SETUP_FAIL	BOTH
LEAF_SETUP_REQ	BOTH

*UP is from q93b to user;
DOWN is from user to q93b

The q93b driver is an M-to-N mux STREAMS driver. Multiple application programs can be plumbed above the driver, and multiple physical interfaces can be connected below q93b. Applications can access any or all of the physical interfaces, and messages received on the physical interfaces may be directed to any of the applications. In order to direct messages through the q93b driver, messages from applications must include a physical interface name to identify the outgoing interface, and a SAP to identify the application to which the message should be directed on the receiving host.

Messages sent to q93b by applications should be sent in the format illustrated in FIGURE 1-2; kernel applications should use `put(9f)` to send the mblocks shown, and user applications should send two corresponding strbufs using `putmsg(2)`.

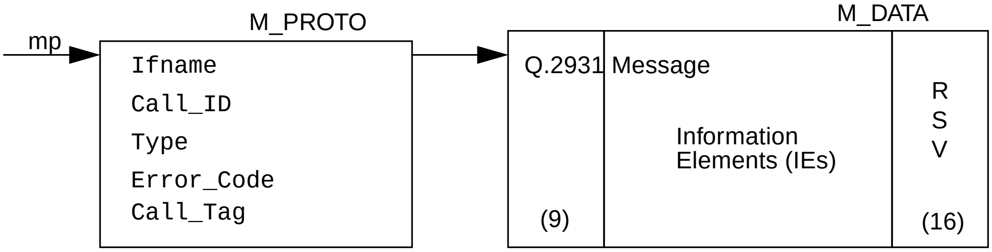

FIGURE 1-2 Message Format

TABLE 1-2 Fields in the M_PROTO mblock

Message	**Explanation**
`Ifname`	A null-terminated string containing the device name
`Call_ID`	A unique number from q93b per interface.
`Type`	The same as the Q.2931 message type except there is a local Non-Q.2931 message type SETUP_ACK. The SETUP_ACK message is used to provide the Call_ID to the user.
`Error_Code`	The error returned from q93b when an erroneous message is received from the user. The exact same mblock chain shall be returned to the user with the Error_Code field set. The user must always clear this field
`Call_Tag`	A number assigned by the calling application layer to a SETUP message. When a SETUP_ACK is received from q93b, the Call_ID has been set; the Call_Tag field can be used to identify the acknowledgment (ack) with the original request. From that point on, the Call_ID value should be used to identify the call.

The structure that is included in the M_PROTO mblock is defined as the `qcc_hdr_t` structure in the `<atm/qcctypes.h>` header file. In the second mblock, the application should leave the Q.2931 header portion (9 bytes) of the Q.2931 message blank; this information is filled in by the q93b driver. The application should also reserve 16 bytes at the end of the second mblock for the layer 2 (Q.SAAL) protocol performance. The qcc functions can be used to create messages in this format.

The following sections give a brief overview of Q.2931 signalling procedures, from the perspective of an application using the SunATM API. For more details on the procedures, refer to the ATM Forum's User Network Interface Specification, version 3.0, 3.1, or 4.0. For further information on the qcc functions, which are outlined in TABLE 1-3, see the appropriate man pages in Section 3 (for user applications) or section 9F (for kernel applications). The man pages can be accessed under the function group name, or any specific function name. For example, the man page which documents the qcc_bld_* function group may be accessed by one of the following at a command prompt: **man qcc_bld**, **man qcc_bld_setup**, or **man qcc_bld_connect**. The message flow during typical call setup and tear down is diagrammed in FIGURE 1-3.

TABLE 1-3 qcc Functions

Name	Functionality	Input	Output
qcc_bld_*	Creates and encodes a message; enables customization of a limited set of values, depending on the message type. Configurable values are passed in as parameters.	Parameter values	Encoded Q.2931 message (in the format shown in FIGURE 1-2)
qcc_parse_*	Extracts a defined set of values from an encoded message	Encoded Q.2931 message (in the format shown in FIGURE 1-2)	Parameter values
qcc_len_*	Returns the maximum length of the buffer that should be allocated for the second strbuf in a Q.2931 message. Only applicable to user space applications; the kernel API allocates the buffers inside the qcc_bld/qcc_pack functions.	none	Maximum length of the message.
qcc_create_*	Creates a message structure with the required values set. The structure can then be further customized using qcc_set_ie.	Default parameter values	Message structure (defined in <atm/qcctypes.h>)
qcc_set_ie	Updates or inserts values for an information element into a message structure.	Message structure and IE structure (defined in <atm/qcctypes.h>)	Updated message structure
qcc_pack_*	Takes a message structure and encodes it into an actual Q.2931 message, consisting of the two mblks (or strbufs) illustrated in FIGURE 1-2.	Message structure (defined in <atm/qcctypes.h>)	Encoded Q.2931 message (in the format shown in FIGURE 1-2)
qcc_unpack_*	The reverse of qcc_pack_*: takes an encoded message and decodes the data into a message structure.	Encoded Q.2931 message (in the format shown in FIGURE 1-2)	Message structure (defined in <atm/qcctypes.h>)
qcc_get_ie	Extracts a single information element structure from a message structure.	Message structure and empty IE structure (defined in <atm/qcctypes.h>)	Updated IE structure

Call Setup

When the user decides to make a call, the user sends a SETUP message down to q93b and waits for a SETUP_ACK from q93b. The SETUP message should include a Broadband Higher Layer Information (BHLI) information element which contains a four-octet SAP identified as User Specific Information. The SAP is used to identify the application to which the message should be directed by q93b on the receiving host. After receiving a SETUP_ACK with a 0 error field, the user waits for either a CALL_PROCEEDING, ALERTING, CONNECT, or RELEASE_COMPLETE message from q93b (all other messages are ignored by q93b). After the CONNECT message is received, the user can use the virtual channel.

When the user receives a SETUP message from q93b, the user responds with either a CALL_PROCEEDING, ALERTING, CONNECT, or RELEASE_COMPLETE message to q93b. After the CONNECT_ACK message is received, the user can use the virtual channel.

Release Procedure

To clear an active call or a call in progress, the user should send a RELEASE message down to q93b and wait for a RELEASE_COMPLETE from q93b. Any time the user receives a RELEASE_COMPLETE message from q93b, the user releases the virtual channel if the call is active or in progress.

q93b never sends a RELEASE message to the user; it will always send a RELEASE_COMPLETE. The user only sends the RELEASE_COMPLETE message when rejecting a call in response to a SETUP message from q93b. At any other time, to reject or tear down a call, the user sends a RELEASE message to q93b.

Exception Conditions

If for any reason q93b cannot process a SETUP message received from a user, the SETUP_ACK is returned with an error value set, and call setup is not continued. The error value will be one of the cause codes specified in the ATM Forum UNI standard.

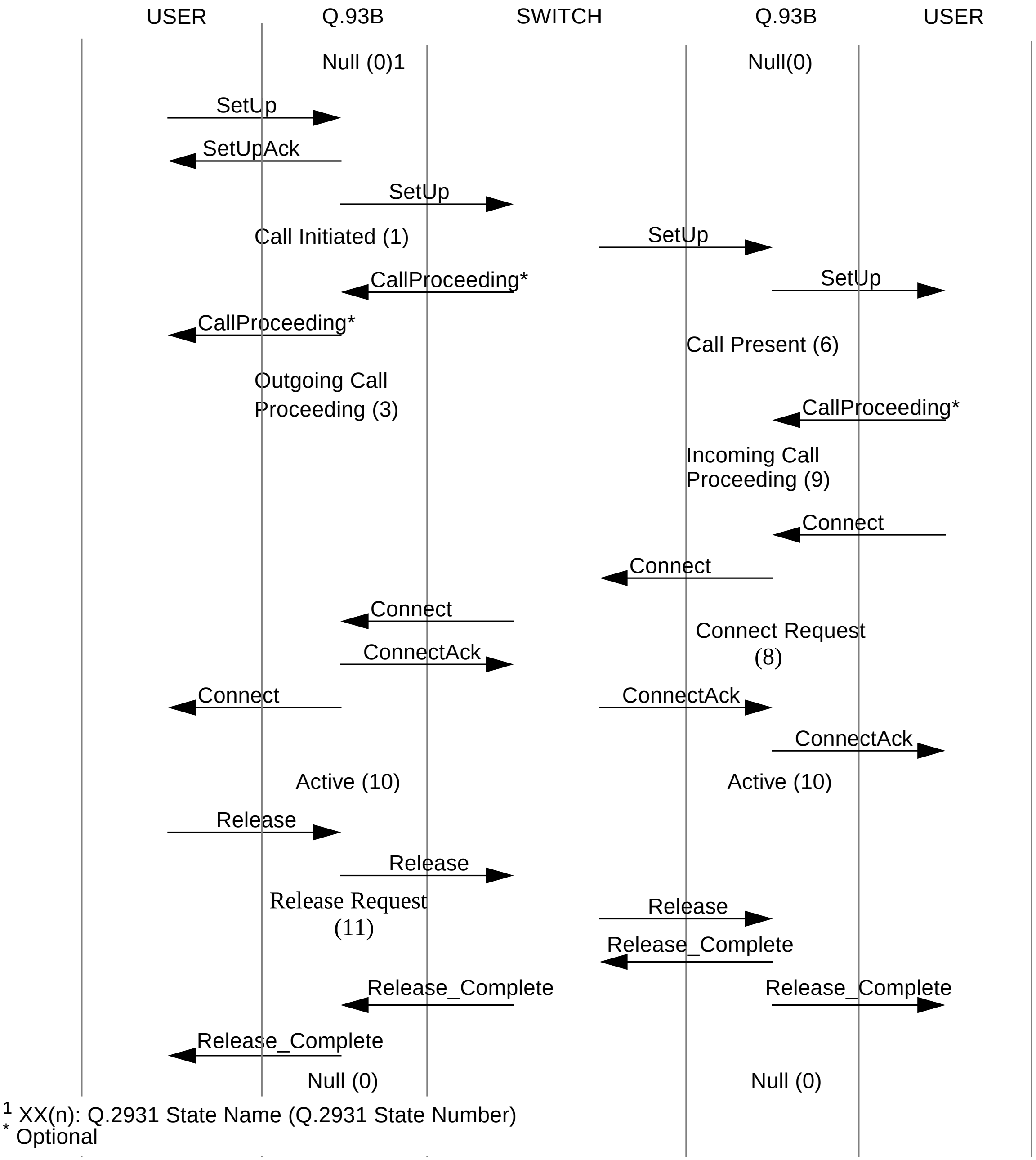

FIGURE 1-3 Normal Call Setup and Tear Down

Connecting, Sending, and Receiving Data with the ATM Device Driver

Connecting to the ATM device driver involves several steps, which include several ioctl calls. In order to create a more standardized interface for user space applications, a set of `atm_util` functions is available to application writers. An overview of those functions is provided in TABLE 1-4. For more detailed information, refer to the `atm_util(3)` man page. The `ba(7)` man page contains a more detailed discussion of the driver-supported IOCTLs.

TABLE 1-4 `atm_util` Function Overview

Name	Functionality	Kernel Equivalent
`atm_open`	Open a stream to the ATM device driver	Must be done by a user space setup program
`atm_close`	Close a stream to the ATM device driver	Must be done by a user space setup program
`atm_attach`	Attach to a physical interface	Must be done by a user space setup program
`atm_detach`	Detach from a physical interface	Must be done by a user space setup program
`atm_bind`	Bind to a Service Access Point	send DL_BIND_REQ
`atm_unbind`	Unbind from a Service Access Point	send DL_UNBIND_REQ
`atm_setraw`	Set the encapsulation mode to raw	Send DLIOCRAW
`atm_add_vpci`	Associate a vpci with this connection	A_ADDVC ioctl
`atm_delete_vpci`	Dissociate a vpci from this connection	A_DELVC ioctl
`atm_allocate_bw`	Allocate constant bit rate bandwidth for this connection	A_ALLOCBW ioctl
`atm_allocate_cbr_bw`	Allocate constant bit rate bandwidth with more granularity than `atm_allocate_bw`	A_ALLOCBW_CBR ioctl
`atm_allocate_vbr_bw`	Allocate variable bit rate bandwidth	A_ALLOCBW_VBR ioctl
`atm_release_bw`	Release previously allocated bandwidth	A_RELSE_BW ioctl

Note – In the following discussion, the user space function names are used. Refer to TABLE 1-4 for the corresponding kernel space function or ioctl.

To establish a data path, the application must first open the ATM driver and attach to a specific physical interface using `atm_open()` and `atm_attach()`. Next, the connection should be associated with one or more VC(s), using `atm_add_vpci()`. If a call has been established using Q.2931 signalling, the vpci provided to `atm_add_vpci()` should be the vpci that was included in the Q.2931 signalling messages received while establishing the call.

An encapsulation method must also be selected. The SunATM device driver supports raw (null) and DLPI encapsulation. Messages sent in raw mode are sent as data only, with just a four-byte vpci as a header; DLPI mode messages are LLC-encapsulated. By default, a connection is in DLPI mode; to change the encapsulation to raw, DLIOCRAW should be set using `atm_setraw()`. The remaining steps depend on the encapsulation mode selected.

Raw Mode Connections

If raw mode is chosen, the only remaining configuration step is to allocate an amount of bandwidth for the use of this connection, using `atm_allocate_bw()`, `atm_allocate_cbr_bw()`, or `atm_allocate_vbr_bw()`.

From the perspective of the application/driver interface, raw mode implies that only a single message buffer (pointed to by dataptr in `putmsg(2)`) should be sent to the driver, containing a 4-byte vpci followed by the data. When a message is received on a vpci running in raw mode, it will be directed to an application based on the vpci. When sending a received message up to the application, the driver will strip the 4-byte vpci from the message if the application has not set DLIOCRAW with a call to atm_setraw; if DLIOCRAW has been set, the 4-byte vpci will be included in the message sent up to the application.

DLPI Encapsulated Connections

If DLPI mode is chosen, a SAP must be associated with the connection using `atm_bind()`. Optionally, a specific amount of bandwidth may be allocated for the connection using `atm_allocate_bw()`, `atm_allocate_cbr_bw()`, or `atm_allocate_vbr_bw()`. If bandwidth is not explicitly allocated, IP’s bandwidth (which includes all available unallocated bandwidth) will be shared by the connection.

DLPI mode implies that two message buffers will be sent to the driver. The first, pointed to by ctlptr in `putmsg(3)`, contains the dlpi message type, which is `dl_unitdata_req` for transmit and `dl_unitdata_ind` for receive. The vpci is included in this buffer as well; the format for the buffer is defined in the header file `<sys/dlpi.h>`. The second buffer, pointed to by dataptr in `putmsg(3)`, contains the data. When the driver receives the two buffers from the application, it will remove the first buffer, add a LLC header containing the SAP which has been bound to this stream to the data buffer, and transmit it. On receive, the LLC header is stripped, the control buffer is added with the DLPI header, and the two buffers are sent up to the application indicated by the SAP in the LLC header.

CHAPTER 2

C Library Functions

The man pages in this chapter describe the C library functions found in the SunATM software. Function declarations can be obtained from the `#include` files indicated on each man page.

TABLE 2-1 C Library Functions

Man Page	Description	Page Number
`atm_util(3)`	SunATM driver utilities, including: `atm_add_vpci(3)`, `atm_allocate_bw(3)`, `atm_allocate_cbr_bw(3)`, `atm_allocate_vbr_bw(3)`, `atm_attach(3)`, `atm_bind(3)`, `atm_close(3)`, `atm_delete_vpci(3)`, `atm_detach(3)`, `atm_open(3)`, `atm_release_bw(3)`, `atm_setraw(3)`, `atm_unbind(3)`	page 19
`qcc_bld(3)`	Build Q.2931 messages, with these commands: `qcc_bld_add_party(3)`, `qcc_bld_add_party_ack(3)`, `qcc_bld_add_party_ack_datalen(3)`,	page 28

TABLE 2-1 C Library Functions *(Continued)*

Man Page	Description	Page Number
	qcc_bld_add_party_datalen(3),	
	qcc_bld_add_party_reject(3),	
	qcc_bld_add_party_reject_datalen(3),	
	qcc_bld_call_proceeding(3),	
	qcc_bld_call_proceeding_datalen(3),	
	qcc_bld_connect(3),	
	qcc_bld_connect_ack_datalen(3),	
	qcc_bld_connect_datalen(3),	
	qcc_bld_drop_party(3),	
	qcc_bld_drop_party_ack(3),	
	qcc_bld_drop_party_ack_datalen(3),	
	qcc_bld_drop_party_datalen(3),	
	qcc_bld_release(3),	
	qcc_bld_release_complete(3),	
	qcc_bld_release_complete_datalen(3),	
	qcc_bld_release_datalen(3),	
	qcc_bld_restart(3),	
	qcc_bld_restart_ack(3),	
	qcc_bld_restart_ack_datalen(3),	
	qcc_bld_restart_datalen(3),	
	qcc_bld_setup(3),	
	qcc_bld_setup_datalen(3),	
	qcc_bld_status(3),	
	qcc_bld_status_datalen(3),	
	qcc_bld_status_enquiry(3),	
	qcc_bld_status_enquiry_datalen(3),	
qcc_create(3)	Create Q.2931 message structures, with these commands:	page 36
	qcc_create_add_party(3),	
	qcc_create_add_party_ack(3),	
	qcc_create_add_party_reject(3),	

TABLE 2-1 C Library Functions *(Continued)*

Man Page	Description	Page Number
	`qcc_create_call_proceeding(3)`, `qcc_create_connect(3)`, `qcc_create_connect_ack(3)`, `qcc_create_drop_party(3)`, `qcc_create_drop_party_ack(3)`, `qcc_create_release(3)`, `qcc_create_release_complete(3)`, `qcc_create_restart(3)`, `qcc_create_restart_ack(3)`, `qcc_create_setup(3)`, `qcc_create_status(3)`, `qcc_create_status_enq(3)`	
`qcc_len(3)`	Get length of Q.2931 messages, with these commands: `qcc_bld_add_party(3)`, `qcc_bld_add_party_ack(3)`, `qcc_bld_add_party_ack_datalen(3)`, `qcc_bld_add_party_datalen(3)`, `qcc_bld_add_party_reject(3)`, `qcc_bld_add_party_reject_datalen(3)`, `qcc_bld_call_proceeding(3)`, `qcc_bld_call_proceeding_datalen(3)`, `qcc_bld_connect(3)`, `qcc_bld_connect_ack_datalen(3)`, `qcc_bld_connect_datalen(3)`, `qcc_bld_drop_party(3)`, `qcc_bld_drop_party_ack(3)`, `qcc_bld_drop_party_ack_datalen(3)`, `qcc_bld_drop_party_datalen(3)`, `qcc_bld_release(3)`, `qcc_bld_release_complete(3)`,	page 45

TABLE 2-1 C Library Functions *(Continued)*

Man Page	Description	Page Number
	qcc_bld_release_complete_datalen(3), qcc_bld_release_datalen(3), qcc_bld_restart(3), qcc_bld_restart_ack(3), qcc_bld_restart_ack_datalen(3), qcc_bld_restart_datalen(3), qcc_bld_setup(3), qcc_bld_setup_datalen(3), qcc_bld_status(3), qcc_bld_status_datalen(3), qcc_bld_status_enquiry(3), qcc_bld_status_enquiry_datalen(3), qcc_ctl_len(3), qcc_len(3), qcc_max_bld_datalen(3)	
qcc_pack(3)	Encode Q.2931 message structure information and pack into streams buffers, with these commands: qcc_pack_add_party(3), qcc_pack_add_party_ack(3), qcc_pack_add_party_reject(3), qcc_pack_call_proceeding(3), qcc_pack_connect(3), qcc_pack_connect_ack(3), qcc_pack_drop_party(3), qcc_pack_drop_party_ack(3), qcc_pack_release(3), qcc_pack_release_complete(3), qcc_pack_restart(3), qcc_pack_restart_ack(3), qcc_pack_setup(3), qcc_pack_status(3),	page 48

TABLE 2-1 C Library Functions *(Continued)*

Man Page	Description	Page Number
	qcc_pack_status_enq(3)	
qcc_parse(3)	Parse Q.2931 messages, including: qcc_parse_add_party(3), qcc_parse_add_party_ack(3), qcc_parse_add_party_reject(3), qcc_parse_call_proceeding(3), qcc_parse_connect(3), qcc_parse_drop_party(3), qcc_parse_drop_party_ack(3), qcc_parse_release(3), qcc_parse_release_complete(3), qcc_parse_restart(3), qcc_parse_restart_ack(3), qcc_parse_setup(3), qcc_parse_status(3), qcc_parse_status_enquiry(3), qcc_get_hdr(3)	page 52
qcc_set_ie(3)	Add or update Information Elements in a Q.2931 message structure	page 60
qcc_unpack(3)	Decode Q.2931 messages and unpack into message structures, with these commands: qcc_unpack(3), qcc_unpack_add_party(3), qcc_unpack_add_party_ack(3), qcc_unpack_add_party_reject(3), qcc_unpack_call_proceeding(3), qcc_unpack_connect(3), qcc_unpack_connect_ack(3), qcc_unpack_drop_party(3), qcc_unpack_drop_party_ack(3), qcc_unpack_release(3),	page 66

TABLE 2-1 C Library Functions *(Continued)*

Man Page	Description	Page Number
	qcc_unpack_release_complete(3), qcc_unpack_restart(3), qcc_unpack_restart_ack(3), qcc_unpack_setup(3), qcc_unpack_status(3), qcc_unpack_status_enq(3)	
qcc_util(3)	Functional interfaces to q93b driver ioctls, including: q_ioc_bind, q_ioc_bind_lijid, q_ioc_unbind_lijid	page 71

atm_util(3)

CODE EXAMPLE 2-1 atm_util(3) Man Page

```
atm_util(3)            C Library Functions            atm_util(3)



NAME
     atm_util,  atm_open,  atm_close,   atm_attach,   atm_detach,
     atm_bind,      atm_unbind,     atm_setraw,     atm_add_vpci,
     atm_delete_vpci,    atm_allocate_bw,    atm_allocate_cbr_bw,
     atm_allocate_vbr_bw,  atm_release_bw - Sun ATM driver utili-
     ties

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/atm.h>

     int atm_open(register char *interface);

     int atm_close(int fd);

     int atm_attach(int fd, u_long ppa, int timeout);

     int atm_detach(int fd, int timeout);

     int atm_bind(int fd, u_long sap, int timeout);

     int atm_unbind(int fd, int timeout);

     int atm_setraw(int fd);

     int atm_add_vpci(int fd, vci_t vpci, int encap,
                      int buf_type);

     int atm_delete_vpci(int fd, vci_t vpci);

     int atm_allocate_bw(int fd, int bw);

     int atm_allocate_cbr_bw(int fd, int bw);

     int atm_allocate_vbr_bw(int fd, int peakbw, int avgbw,
```

CODE EXAMPLE 2-1 atm_util(3) Man Page *(Continued)*

```
                             int maxburst, int priority);

     int atm_release_bw(int fd);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the  SUNWatma  package included with a SunATM adapter board.
     The libatm.a library, which is located in /opt/SUNWatm/lib, must
     be included at compile time as indicated in the synopsis.

DESCRIPTION
     These utilities perform various  operations  on  the  SunATM
     device driver, ba.  They may be used by application programs
     that need to transmit and receive data over an  ATM  connec-
     tion to set up a data stream to the ATM driver.

     Data may be transmitted over a vc connection in one  of  two
     modes: raw mode, or dlpi mode. The default is dlpi mode. Raw
     mode may be requested by  sending  down  a  DLIOCRAW  ioctl,
     which is accomplished with a call to atm_setraw().  The mode
     chosen defines the format in which data should  be  sent  to
     the driver.

     Raw mode implies that only a single mblock will be  sent  to
     the  driver,  containing  a  four-byte  vpci followed by the
     data.  When a message is received on a vpci running  in  raw
     mode, the four-byte vpci will be sent up with the data.

     DLPI mode implies that two  mblocks  will  be  sent  to  the
     driver.   The first, of type M_PROTO, contains the dlpi mes-
     sage  type,  which  is  dl_unitdata_req  for  transmit   and
     dl_unitdata_ind  for  receive.  The vpci is included in this
     mblock as well. The dl_unitdata_req and dl_unitdata_ind header
     formats are defined  in  the  header  file <sys/dlpi.h>.
     The second mblock is of type M_DATA and con-
     tains the message. When the driver gets a message of this
     type  from the upper layer, it will remove the first mblock,and
     transmit the message.  On receive, the M_PROTO  mblock is added,
     and the two-mblock structure is sent up to the user.

     A method of encapsulation must also be chosen; the method of
     encapsulation  is specified when the VC is associated with a
     stream (using the A_ADDVC ioctl or the atm_add_vpci()  func-
```

CODE EXAMPLE 2-1 atm_util(3) Man Page *(Continued)*

```
tion  call).  Currently, null and LLC encapsulation are sup-
ported.  Null encapsulation implies that a message  consists
only  of  data  preceded  by a four-byte vpci.  This type of
encapsulation is most commonly  used  with  raw  mode.   LLC
encapsulation  implies that an LLC header precedes the data.
This  header  will  include  the  SAP  associated  with  the
application's  stream  (using the atm_bind() function call).
This type of encapsulation is typically used with dlpi  mode
traffic.

For LLC-encapsulated traffic, the driver will  automatically
add  the  LLC header on transmit if the stream is running in
dlpi mode.  The driver will also strip the LLC  header  from
incoming  traffic  before  sending it up a dlpi mode stream.
In raw mode, however, the driver does not modify the packets
at  all; this includes the LLC header.  Thus, an application
using raw mode and LLC encapsulation must  include  its  own
LLC  headers  on transmit and will receive data with the LLC
header intact.

Received packets are directed to application streams by  the
driver  based  on the type of encapsulation.  If a packet is
null-encapsulated, it will be sent up the stream  associated
with the vpci on which the packet was received.  If a packet
is LLC-encapsulated, it will be sent to the stream which has
bound (using atm_bind()) the SAP found in the LLC header.

NOTE: If the application is running  in  user  space  rather
than kernel space, the M_PROTO and M_DATA mblocks correspond
to the ctl and data buffers, respectively, which are  passed
into putmsg(2) or received from getmsg(2).


atm_open() opens a stream to the  physical  interface  (i.e.
ba0,  ba1,  etc.)  passed  in as a null-terminated string in
interface.  On success, the file  descriptor  (  >  0  )  is
returned.

atm_close() closes the stream specified by its file descrip-
tor, fd.

atm_attach() associates a physical point of attachment, ppa,
with  an  opened ba device specified by its file descriptor,
fd.  The ppa is usually defined as  the  physical  interface
number (0 for ba0, 1 for ba1, etc.).  timeout may optionally
be used to specify an amount of time in milliseconds to wait
```

CODE EXAMPLE 2-1 atm_util(3) Man Page *(Continued)*

```
     for  the function to complete.  The function will fail if it
     does not complete in the specified amount of time.  Possible
     values for timeout are -1, which blocks until completion, 0,
     which returns immediately, or a number greater than 0  which
     specifies a number of milliseconds to wait.  This value will
     be rounded up to an implementation-dependent minimum  value,
     which is currently at  approximately 100 ms.

     atm_detach() detaches  the  stream  specified  by  its  file
     descriptor  fd  from  its  ppa.   Values of timeout apply as
     described in atm_attach().

     atm_allocate_bw() specifies a constant  bit  rate  bandwidth
     amount  in  megabits per second (Mbps), passed in as bw. The
     amount  of  bandwidth  specified  will  be   allocated   for
     transmitting  data  from  the  stream identified by the file
     descriptor fd.  All unallocated bandwidth is assigned to  IP
     and LLC-encapsulated traffic. This step is not necessary if a
     stream is only to be used to receive data; nor is it  neces-
     sary to allocate bandwidth for a stream which is sending LLC-.
     encapsulated traffic.
     By default, LLC-encapsulated traffic shares  all unallocated
     bandwidth  with  IP.  See  the table below for the amount of
     bandwidth available to be allocated by the  user.  Bandwidth
     may   be   allocated   to   a   finer   granularity   using
     atm_allocate_cbr_bw().

     atm_allocate_cbr_bw() specifies an amount  of  constant  bit
     rate  bandwidth  in  units of 64 kilobits per second (Kbps),
     passed in as bw. The amount of bandwidth specified  will  be
     allocated  for  transmitting data from the stream identified
     by the file descriptor  fd.  All  unallocated  bandwidth  is
     assigned to IP  and LLP-encapsulated traffic. Allocation  of
     bandwidth is not necessary if a stream is only to be used to
     receive  data; nor is it necessary to allocate bandwidth for
     a stream running in raw mode. By default, dlpi mode  traffic
     shares  all  unallocated  bandwidth  with IP.  See the table
     below for the amount of bandwidth available to be  allocated
     by  the user. Bandwidth may be allocated with less granular-
     ity   (in   units   of   megabits    per    second)   using
     atm_allocate_bw().

     atm_allocate_vbr_bw() specifies an amount  of  variable  bit
     rate  bandwidth to allocate for the stream identified by the
     file descriptor fd. Variable bit rate traffic is implemented
     by  the  SunATM hardware according to the GCRA (Generic Cell
```

CODE EXAMPLE 2-1 atm_util(3) Man Page *(Continued)*

```
Rate Algorithm) as defined by the ATM Forum UNI 3.0 specifi-
cation.   The  parameters  peakbw and avgbw are passed in in
units of 64 kilobits per second (Kbps),  and  represent  the
Peak  Cell Rate and Sustainable Cell Rate, respectively. The
Sustainable Cell Rate must be available within the bandwidth
parameters  of the hardware, which are described in the fol-
lowing table. The maxburst parameter specifies the number of
cells  which  may  be  sent  back  to  back  on  the  media,
corresponding to the Maximum Burst Size  in  the  UNI  spec.
Finally,  priority  may  be  AVBR_HIGH_PRI  or  AVBR_LO_PRI;
AVBR_HIGH_PRI will always get the requested bandwidth, while
AVBR_LO_PRI  can starve if other users request all available
bandwidth.

Available Bandwidth
--------------------------------------------------------------
| Product              |    SunATM-155    |    SunATM-622    |
|----------------------+------------------+------------------|
| Unit of Measure      |  Mbps  | 64 Kbps |  Mbps  | 64 Kbps |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Total Bandwidth      |  155   |  2480   |  622   |  9952   |
|                      |        |         |        |         |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Cell Header/Phy      |   20   |   320   |   88   |  1408   |
|   Layer Overhead     |        |         |        |         |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Reserved by Software |  0.125 |     2   |  0.125 |     2   |
|                      |        |         |        |         |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Available to User    |134.875 |  2158   |533.875 |  8542   |
|                      |        |         |        |         |
--------------------------------------------------------------

atm_release_bw() releases all bandwidth that has been previ-
ously allocated to the stream identified by fd.

atm_add_vpci() adds the given virtual path connection  iden-
tifier,  vpci,  to  those recognized on the specified stream
(identified by its file descriptor, fd).  The type of encap-
sulation  that is being used on this connection must also be
specified in encap;  the  possible  values  are  NULL_ENCAP,
LLC_ENCAP,  and NLPID_ENCAP, as defined in <atm/atmioctl.h>.
```

CODE EXAMPLE 2-1 `atm_util(3)` Man Page *(Continued)*

```
     Finally, the buffer type  must  be  specified  in  buf_type;
     definitions  may  also  by found in <atm/atmioctl.h> for the
     possible    types    SMALL_BUF_TYPE,    BIG_BUF_TYPE,    and
     HUGE_BUF_TYPE.

     atm_delete_vpci()  deletes  given  virtual  path  connection
     identifier,  vpci,  from the specified stream (identified by
     its file descriptor, fd).

     atm_bind() binds a service access point, sap, to  an  opened
     stream, specified by its file descriptor, fd.  sap values of
     0x800 and 0x806 are reserved for IP and ARP traffic, respec-
     tively;  the  user  shall  not use these values.  The sap is
     used by the driver to direct traffic to upper layers if  LLC
     encapsulation  is  used.   This  function also has a timeout
     parameter; the values of timeout described  in  atm_attach()
     apply in atm_bind() as well.
     atm_unbind() disassociates  a  stream-to-sap  binding.   The
     stream  is  specified by its file descriptor, fd.  Values of
     timeout apply as described in atm_attach().

     atm_setraw() indicates to the driver that the stream  speci-
     fied  by  the  file  descriptor  fd will be transmitting and
     receiving raw data which will be interpreted directly by the
     application at the stream head.  The only header information
     included in messages passed down the stream will be  the  4-
     byte  virtual path connection identifier.  When a message is
     received, the vpci will be used to  direct  the  message  to
     upper layers.

     The ordering of the atm utility function calls is important.
     After  calling  atm_open(),  the order must be atm_attach(),
     followed by atm_add_vpci().  Next, depending on the type  of
     encapsulation   used   on  this  stream,  should  be  either
     atm_bind() for LLC encapsulation (dlpi mode) or atm_setraw()
     for null encapsulation (raw mode). Finally, bandwidth may be
     allocated with a call to atm_alloc_bw(), atm_alloc_cbr_bw(),
     or  atm_alloc_vbr_bw().   All  functions must be called only
     once per interface, with the  exception  of  atm_add_vpci(),
     which  may  be  called  multiple  times  to support multiple
     vpcis.

RETURN VALUES
     All functions return -1 on error.   With  the  exception  of
     atm_open,  which returns the file descriptor on success, all
     functions return 0 on success.
```

CODE EXAMPLE 2-1 atm_util(3) Man Page *(Continued)*

```
EXAMPLES
     The following example opens a stream to ba0 and sets up that
     stream to communicate over vpci 0x100 at 10 Mbits/sec in raw
     mode.

          #include <stdio.h>
          #include <sys/types.h>
          #include <sys/stropts.h>
          #include <sys/errno.h>
          #include <atm/atm.h>

          main()
          {
               char     interface[] = "ba0";
               int      fd;
               int      ppa;
               int      bw = 10;
               int      vpci = 0x100;
               char     ctlbuf[256];
               char     databuf[256];
               struct strbuf   ctl, data;
               ctl.buf = ctlbuf;
               data.buf = databuf;
               ctl.maxlen = data.maxlen = 256;

               ppa = atoi(&interface[strlen (interface) - 1]);
               if ((fd = atm_open(interface)) < 0) {
                    perror("open");
                    exit(-1);
               }
               atm_attach(fd, ppa);

               if (atm_add_vpci(fd, vpci, LLC_ENCAP, BIG_BUF_TYPE) < 0) {
                    perror("atm_add_vpci");
                    exit(-1);
               }
               if (atm_setraw(fd) < 0) {
                    perror("atm_setraw");
                    exit(-1);
               }

               <construct a message to pass down in ctlbuf and databuf>

               if (putmsg(fd, &ctl, &data, 0) < 0) {
                    perror("putmsg");
```

CODE EXAMPLE 2-1 atm_util(3) Man Page *(Continued)*

```
               exit(-1);
          }

     }

The following example opens a stream to ba0 and sets up that
stream  to  communicate over vpci 0x100, using sap 0x100, in
dlpi mode.

     #include <stdio.h>
     #include <sys/types.h>
     #include <sys/stropts.h>
     #include <sys/errno.h>
     #include <sys/dlpi.h>
     #include <atm/atm.h>

     main()
     {
          char     interface[] = "ba0";
          int      fd;
          int      ppa;
          int      vpci = 0x100;
          int      *vpcip;
          int      sap = 0x100;
          char     ctlbuf[256];
          char     databuf[256];
          struct strbuf      ctl, data;
          dl_unitdata_req_t  *dludp;
          ctl.buf = ctlbuf;
          data.buf = databuf;
          ctl.maxlen = data.maxlen = 256;

          ppa = atoi(&interface[strlen (interface) - 1]);
          if ((fd = atm_open(interface)) < 0) {
               perror("open");
               exit(-1);
          }
          atm_attach(fd, ppa);

          if (atm_add_vpci(fd, vpci, LLC_ENCAP, BIG_BUF_TYPE) < 0) {
               perror("atm_add_vpci");
               exit(-1);
          }
          atm_bind(fd, sap);

          <construct the message in databuf>
```

CODE EXAMPLE 2-1 `atm_util(3)` Man Page *(Continued)*

```
               ctllen = sizeof (dl_unitdata_req_t) + 4;
               memset(ctlbuf, 0, ctllen);
               dludp = (dl_unitdata_req_t *) ctlbuf;
               dludp->dlprimitive = DL_UNITDATA_REQ;
               dludp->dl_dest_addr_length = 4;
               dludp->dl_dest_addr_offset = sizeof (dl_unitdata_req_t);
               vpcip = (int *) &ctlbuf[sizeof (dl_unitdata_req_t)];
               *vpcip = vpci;

               if (putmsg(fd, &ctl, &data, 0) < 0) {
                    perror("putmsg");
                    exit(-1);
               }
          }

SEE ALSO
     dlpi(7), ba(7)
```

qcc_bld(3)

CODE EXAMPLE 2-2 qcc_bld(3) Man Page

```
qcc_bld(3)             C Library Functions            qcc_bld(3)



NAME
     qcc_bld,          qcc_bld_setup,           qcc_bld_alerting,
     qcc_bld_call_proceeding,  qcc_bld_connect,  qcc_bld_release,
     qcc_bld_release_complete,                    qcc_bld_status,
     qcc_bld_status_enquiry,   qcc_bld_notify,   qcc_bld_restart,
     qcc_bld_restart_ack,                      qcc_bld_add_party,
     qcc_bld_add_party_ack,               qcc_bld_party_alerting,
     qcc_bld_add_party_reject,                qcc_bld_drop_party,
     qcc_bld_drop_party_ack,             qcc_bld_leaf_setup_fail,
     qcc_bld_leaf_setup_req - build Q.2931 messages

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/types.h>
     #include <atm/qcc.h>

     int qcc_bld_setup(strbuf_t *ctlp, strbuf_t *datap,
           char *ifname, int calltag, int vci,
           int forward_sdusize, int backward_sdusize,
           atm_addr_t *src_addrp, atm_addr_t *dst_addrp,
           int sap, int endpt_ref);

     int qcc_bld_alerting(strbuf_t *ctlp, strbuf_t *datap,
           char *ifname, int callid, int vci, int endpt_ref);

     int qcc_bld_call_proceeding(strbuf_t *ctlp, strbuf_t *datap,
           char *ifname, int callid, int vci, int endpt_ref);

     int qcc_bld_connect(strbuf_t *ctlp, strbuf_t *datap,
           char *ifname, int callid, int vci,
           int forward_sdusize, int backward_sdusize,
           int endpt_ref);

     int qcc_bld_release(strbuf_t *ctlp, strbuf_t *datap,
           char *ifname, int callid, int cause);
```

CODE EXAMPLE 2-2 `qcc_bld(3)` Man Page *(Continued)*

```
int qcc_bld_release_complete(strbuf_t *ctlp,
      strbuf_t *datap, char *ifname, int callid, int cause);

int qcc_bld_status_enquiry(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int endpt_ref);

int qcc_bld_status(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int callstate, int cause,
      int endpt_ref, int endpt_state);

int qcc_bld_notify(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int contentlen,
      u_char *contentp, int endpt_ref);

int qcc_bld_restart(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int vci, int rstall);

int qcc_bld_restart_ack(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int vci, int rstall);

int qcc_bld_add_party(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int forward_sdusize,
      int backward_sdusize, atm_address_t *src_addrp,
      atm_address_t *dst_addrp, int sap, int endpt_ref);

int qcc_bld_add_party_ack(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int endpt_ref);

int qcc_bld_party_alerting(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int endpt_ref);

int qcc_bld_add_party_reject(strbuf_t *ctlp,
      strbuf_t *datap, char *ifname, int callid, int cause,
      int endpt_ref);

int qcc_bld_drop_party(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int cause, int endpt_ref);

int qcc_bld_drop_party_ack(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int cause, int endpt_ref);

int qcc_bld_leaf_setup_fail(strbuf_t *ctlp, strbuf_t *datap,
      char *ifname, int callid, int cause,
      atm_address_t *dst_addrp, int leaf_num);
```

CODE EXAMPLE 2-2 qcc_bld(3) Man Page *(Continued)*

```
     int qcc_bld_leaf_setup_req(strbuf_t *ctlp, strbuf_t *datap,
           char *ifname, int leaftag, atm_address_t *src_addrp,
           atm_address_t *dst_addrp, int lij_callid);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions build the various messages that make up  the
     Q.2931  protocol  which  is used for ATM signalling.  A full
     description of the message format and use can  be  found  in
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0. The messages built will conform to  the  ver-
     sion  of  the  UNI  Specification which is configured on the
     indicated interface. The functions may be used by  processes
     which are running in user space.

     In general, no error checking is performed on the data  that
     is  passed in.  Whatever data is passed in will be placed in
     the message that is built  without  examination.   The  only
     exceptions  to  this  are mentioned in the function descrip-
     tions.

     Each function requires a minimum of 4 parameters:  ctlp  and
     datap, which are pointers to strbuf_t buffers; ifname, which
     is a string containing the physical interface (such as ba0);
     and  an  integer, either calltag or callid, depending on the
     message type.  calltag is used in the setup message only; it
     is a reference number that is assigned by the calling appli-
     cation.  callid  is  used  in  all  other  messages;  it  is
     assigned by the lower layer and will be sent up to the user,
     with the calltag, in the setup_ack message.

     ctlp and datap make up the control and data portions of  the
     constructed message, corresponding to the M_PROTO and M_DATA
     blocks of the message that will be passed  downstream.   The
     buffer  fields  in the structures which ctlp and datap point
     to (ctlp->buf and datap->buf) must be allocated before  cal-
     ling  a  qcc_bld* function; size information may be obtained
     using the qcc_bld_*_datalen()  functions  (see  qcc_len(3)).
```

CODE EXAMPLE 2-2 qcc_bld(3) Man Page *(Continued)*

```
     After  successful  return from a qcc_bld* function, the mes-
     sage may be passed down an open stream using  the  putmsg(2)
     function,  with  ctlp and datap as the buffer parameters for
     putmsg.

     Other parameters for each function depend  on  the  type  of
     information  required for each message type, and are defined
     in the paragraphs describing each function call.

     After a message has been built, the user may  add  IEs  that
     are  not  built into the message; however, the size informa-
     tion returned by the qcc_len functions only includes the IEs
     documented  here.   The user must allocate enough additional
     space and correct the message length  value  in  the  Q.2931
     header if additional IEs are required in the message.

     qcc_bld_setup() constructs a setup message  containing  some
     or  all  of  the following Information Elements: AAL parame-
     ters, ATM  user  cell  rate,  broadband  bearer  capability,
     called  party  number, calling party number, quality of ser-
     vice parameter, and endpoint reference. The user  must  pass
     in  the forward and backward sdu sizes for the AAL parameter
     IE, an ATM address for the destination for the called  party
     number  IE,  and one for itself for the calling party number
     IE (atm_address_t  format  is  defined  in  the  <atm/qcc.h>
     header  file).   The  value  passed  in the sap parameter is
     placed in a broadband higher layer IE. The higher  layer  IE
     indicates  the  sap  to  which  received  messages should be
     directed. If the user passes in a positive vci, a connection
     identifier  IE  will  be  included;  if the user passes in a
     non-negative endpt_ref  value  (0  is  valid),  an  endpoint
     reference  IE  is  included. The endpoint reference IE indi-
     cates that this is a point-to-multipoint call.

     qcc_bld_alerting() is specific to UNI  4.0.   It  builds  an
     alerting message containing a  connection identifier IE if a
     positive vci is passed in, and an endpoint reference IE if a
     non-negative  endpt_ref is passed in.  An endpoint reference
     IE should only appear if the call is  a  point-to-multipoint
     call.  The alerting message is only supported under UNI 4.0.

     qcc_bld_call_proceeding() includes a  connection  identifier
     IE if a positive vci is passed in, and an endpoint reference
     IE if a non-negative endpt_ref is  passed  in.  An  endpoint
     reference  IE  should only appear if the call is a point-to-
     multipoint call.
```

CODE EXAMPLE 2-2 qcc_bld(3) Man Page *(Continued)*

```
     qcc_bld_connect() includes an AAL parameters  IE,  requiring
     the forward_ and backward_sdusize values, a connection iden-
     tifier IE if a positive vci value is passed in, and an  end-
     point  reference  IE  if  a  non-negative endpt_ref value is
     passed in. An endpoint reference IE should  only  appear  if
     the call is a point-to-multipoint call.

     qcc_bld_release() includes a cause IE  for  which  the  user
     must  pass  in  a  cause  value.  The possible values can be
     found in the <atm/qcc.h> header file.  The same is true  for
     qcc_bld_release_complete().

     qcc_bld_status_enquiry() includes only an endpoint reference
     IE  if  a non-negative endpt_ref value is passed in. An end-
     point reference IE should only  appear  if  the  call  is  a
     point-to-multipoint call.

     qcc_bld_status() includes a call  state  IE,  requiring  the
     user pass in the callstate parameter; possible values can be
     found in the <atm/qcc.h> header file.  It  also  includes  a
     cause  IE; the cause value must also be passed in.  Its pos-
     sible values may also be found  in  the  <atm/qcc.h>  header
     file.  Finally,  if  the call is a point-to-multipoint call,
     endpoint reference  and  endpoint  state  IEs  may  also  be
     included;  they  are  included  if  a non-negative endpt_ref
     value is passed in. The endpt_state parameter is used in the
     endpoint  state IE; possible party state values may be found
     in <atm/qcc.h>.

     qcc_bld_notify() is specific to UNI 4.0.  It builds a notify
     message,  including  a notification indicator IE, which con-
     tains a buffer of user-defined information up to  a  maximum
     length of 16 bytes (defined by contentlen and contentp), and
     an endpoint reference IE if a non-negative  endpt_ref  value
     is  passed  in.  An endpoint reference IE should only appear
     if the call is a point-to-multipoint call.  The notify  mes-
     sage is only valid under UNI 4.0.

     qcc_bld_restart() includes a restart indicator IE, which  is
     used to determine whether an individual call or all calls on
     an interface should be restarted.  If rstall is 0, only  the
     call  identified by vci should be restarted; in this case, a
     connection identifier IE will also be included.   If  rstall
     is  non-zero,  all calls will be restarted.  The same format
     applies to the qcc_bld_restart_ack() function.
```

CODE EXAMPLE 2-2 `qcc_bld(3)` Man Page *(Continued)*

```
     qcc_bld_add_party() constructs an add party  message  for  a
     point-to-multipoint  call. The message constructed will con-
     tain an AAL parameters IE, which includes the  forward_  and
     backward_sdusize  parameters,  a  calling  party  number IE,
     which includes the value pointed to by src_addrp,  a  called
     party  number  IE,  which  includes  the value pointed to by
     dst_addrp, a broadband higher layer  information  IE,  which
     includes  the  sap  parameter, and an endpoint reference IE,
     which includes the endpt_ref parameter. The sap value in the
     broadband  higher  layer information IE indicates the sap to
     which the message should be passed by the receiving host.

     qcc_bld_add_party_ack() constructs an add party ack  message
     which  includes  an  endpoint  reference  IE,  for which the
     endpt_ref parameter is required.

     qcc_bld_party_alerting() is specific to UNI 4.0.  It  builds
     a  party  alerting message, containing an endpoint reference
     IE, for which the endpt_ref parameter is required.

     qcc_bld_add_party_reject() includes a cause  IE,  containing
     the  cause value passed in. The possible cause values may be
     found in the <atm/qcc.h> header file. An endpoint  reference
     IE is also included, which requires the endpt_ref parameter.

     qcc_bld_drop_party() constructs a drop  party  message.  The
     message  constructed will contain two IEs: a cause IE, which
     requires the cause parameter, and an endpoint reference  IE,
     which  requires  the  endpt_ref  parameter.  Possible  cause
     values may be found in the header file <atm/qcc.h>.

     qcc_bld_drop_party_ack() contains an endpoint reference  IE,
     requiring  the  endpt_ref parameter, and optionally, a cause
     IE. The cause IE will be included if a positive cause  value
     is  passed  in.  Possible  cause  values may be found in the
     <atm/qcc.h> header file.

     qcc_bld_leaf_setup_fail() is specific to UNI 4.0.   It  con-
     tains a cause IE if a non-negative cause value is passed in;
     a called number IE if a non-null dst_addrp is passed in; and
     a  leaf  number  IE,  for  which  the  leaf_num parameter is
     required.  This message type is only valid under UNI 4.0.

     qcc_bld_leaf_setup_req() is specific to UNI  4.0.   It  con-
     tains  Calling  Number  and  Called  Number  IEs if non-null
```

CODE EXAMPLE 2-2 qcc_bld(3) Man Page *(Continued)*

```
     src_addrp and dst_addrp are passed in, respectively; it also
     contains  a leaf initiated join call identifier IE for which
     lij_callid is required, and a  leaf  number  IE.   The  leaf
     number  is  assigned  by  the q93b driver.  Because the leaf
     number is assigned by the q93b driver, a  mechanism  similar
     to  that  used  in  the setup and setup_ack messages is used
     with the leaf number: the  user  must  provide  a  'leaftag'
     parameter  in the call to qcc_bld_leaf_setup_req(); this tag
     is inserted in the calltag field of the  qcc  header.   When
     the  message  is received and accepted by the q93b driver, a
     leaf_setup_ack message  is  returned,  containing  both  the
     leaftag,  in  the  calltag  field of the qcc header, and the
     driver-assigned leaf number,  in  the  callref  field.   The
     leaf_setup_req and leaf_setup_ack messages are the only mes-
     sages which will not contain a call reference value  in  the
     callref  field; this is because the messages are not tied to
     a specific call.  This message, and the leaf-initiated  join
     functionality, are only supported under UNI 4.0.

RETURN VALUES
     All functions return 0 on success and -1 on error.

EXAMPLES
     The following code fragment builds a setup message and sends
     it downstream.

          #include <atm/limits.h>
          #include <atm/qcc.h>

          char    ifname[QCC_MAX_IFNAME_LEN] = "ba0";
          int     calltag = 0x1234;
          int     vci = 0x100;
          int     forward_sdusize = 0x2378;
          int     backward_sdusize = 0x2378;
          int     sap = 0x100;

          atm_addr_t     src_addr = {
               0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
               0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
               0x08, 0x00, 0x20, 0x1a, 0xe1, 0x53, 0x00
          };

          atm_addr_t     dst_addr = {
               0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
               0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
               0x08, 0x00, 0x20, 0x1a, 0xb6, 0xb9, 0x00
```

CODE EXAMPLE 2-2 `qcc_bld(3)` Man Page *(Continued)*

```
          };

          struct strbuf  ctl, data;
          char           ctlbuf[QCC_MAX_CTL_LEN];
          char           databuf[QCC_MAX_DATA_LEN];

          ctl.buf = ctlbuf;
          data.buf = databuf;
          ctl.maxlen = QCC_MAX_CTL_LEN;
          data.maxlen = QCC_MAX_DATA_LEN;

          if ((qcc_bld_setup(&ctl, &data, ifname, calltag, vci,
                              forward_sdusize, backward_sdusize,
                              &src_addr, &dst_addr, sap, -1)) < 0) {
               printf("qcc_bld_setup failed\n");
               exit (-1);
          }

          if (putmsg(fd, &ctl, &data, 0) < 0) {
               perror("putmsg");
               exit (-1);
          }

SEE ALSO
     qcc_len(3), qcc_parse(3), qcc_util(3), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.  These message types, if sent on an interface
     configured for UNI 3.0 or 3.1, will be discarded by the q93b
     driver  and  will  not  be  sent out to the network.  The UNI
     4.0-specific messages are Alerting, Notify, Party  Alerting,
     Leaf  Setup Fail, and Leaf Setup Request, and are identified
     in the applicable function descriptions.
```

qcc_create(3)

CODE EXAMPLE 2-3 qcc_create(3) Man Page

```
qcc_create(3)          C Library Functions          qcc_create(3)



NAME
     qcc_create,      qcc_create_setup,      qcc_create_alerting,
     qcc_create_call_proceeding,             qcc_create_connect,
     qcc_create_connect_ack,                 qcc_create_release,
     qcc_create_release_complete,             qcc_create_status,
     qcc_create_status_enq,                   qcc_create_notify,
     qcc_create_restart,                 qcc_create_restart_ack,
     qcc_create_add_party,             qcc_create_add_party_ack,
     qcc_create_party_alerting,      qcc_create_add_party_reject,
     qcc_create_drop_party,            qcc_create_drop_party_ack,
     qcc_create_leaf_setup_fail,    qcc_create_leaf_setup_req   -
     create Q.2931 message structures

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/qcc.h>
     #include <atm/qcctypes.h>

     int qcc_create_setup(qcc_setup_t *msgp, char *ifname,
           int calltag, atm_address_t *dst_addrp);

     int qcc_create_alerting(qcc_alerting_t *msgp, char *ifname,
           int callid);

     int qcc_create_call_proceeding(qcc_call_proc_t *msgp,
           char *ifname, int callid);

     int qcc_create_connect(qcc_connect_t *msgp, char *ifname,
           int callid);

     int qcc_create_connect_ack(qcc_connect_ack_t *msgp,
           char *ifname, int callid);

     int qcc_create_release(qcc_release_t *msgp, char *ifname,
           int callid, int cause);
```

CODE EXAMPLE 2-3 `qcc_create(3)` Man Page *(Continued)*

```
int qcc_create_release_complete(qcc_release_complete_t *
      msgp, char *ifname, int callid);

int qcc_create_status_enq(qcc_status_enq_t *msgp,
      char *ifname, int callid);

int qcc_create_status(qcc_status_t *msgp, char *ifname,
      int callid, int callstate, int cause);

int qcc_create_notify(qcc_notify_t *msgp, char *ifname,
      int callid, int contentlen, u_char *contentp);

int qcc_create_restart(qcc_restart_t *msgp, char *ifname,
      int callid, int indicator, int vci);

int qcc_create_restart_ack(qcc_restart_ack_t *msgp,
      char *ifname, int callid, int indicator, int vci);

int qcc_create_add_party(qcc_add_party_t *msgp,
      char *ifname, int callid, atm_address_t *dst_addrp,
      int endpt_ref);

int qcc_create_add_party_ack(qcc_add_party_ack_t *msgp,
      char *ifname, int callid, int endpt_ref);

int qcc_create_party_alerting(qcc_party_alerting_t *msgp,
      char *ifname, int callid, int endpt_ref);

int qcc_create_add_party_reject(qcc_add_party_reject_t *
      msgp, char *ifname, int callid, int cause,
      int endpt_ref);

int qcc_create_drop_party(qcc_drop_party_t *msgp,
      char *ifname, int callid, int cause, int endpt_ref);

int qcc_create_drop_party_ack(qcc_drop_party_ack_t *msgp,
      char *ifname, int callid, int endpt_ref);

int qcc_create_leaf_setup_fail(qcc_leaf_setup_fail_t *msgp,
      char *ifname, int callid, int cause,
      atm_address_t *dst_addrp, int leaf_num);

int qcc_create_leaf_setup_req(qcc_leaf_setup_req_t *msgp,
      char *ifname, int leaftag, atm_address_t *src_addrp,
      atm_address_t *dst_addrp, int lij_callid);
```

CODE EXAMPLE 2-3 qcc_create(3) Man Page *(Continued)*

```
MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions create message structures  representing  the
     various  messages that make up the Q.2931 protocol, which is
     used for ATM signalling.  A full description of the  message
     format  and use can be found in the ATM Forum's User Network
     Interface Specification, V3.0, V3.1, or V4.0. The content of
     the  created  message structures will conform to the version
     of the UNI Specification which is configured  on  the  indi-
     cated  interface.  The  functions  may  be used by processes
     which are running in user space.

     After a message  structure  has  been  created,  non-default
     Information  Elements (IEs) may be added or existing IEs may
     be changed using the qcc_set_ie(3) function. When  the  mes-
     sage   structure   has   been   completely   specified,  the
     corresponding  qcc_pack(3)  function  should  be  called  to
     translate  the  message  structure  into the correct encoded
     format, contained in streams buffers which may be passed  to
     the putmsg(2) function.

     In general, no error checking is performed on the data  that
     is  passed in.  Whatever data is passed in will be placed in
     the message that is built  without  examination.   The  only
     exceptions  to  this  are mentioned in the function descrip-
     tions.

     Each function requires a  minimum  of  3  parameters:  msgp,
     which  is  a  pointer  to  the appropriate message structure
     type; ifname, which is  a  string  containing  the  physical
     interface  (such  as ba0); and an integer, either calltag or
     callid, depending on the message type.  calltag is  used  in
     the  setup  message  only;  it is a reference number that is
     assigned by the calling application.  callid is used in  all
     other  messages;  it is assigned by the lower layer and will
     be sent up to the user, with the calltag, in  the  setup_ack
     message.
```

CODE EXAMPLE 2-3 qcc_create(3) Man Page *(Continued)*

```
The structure to which msgp points must be allocated by  the
calling  user.  There is a unique structure for each message
type;   the    message    structures    are    defined    in
<atm/qcctypes.h>.

Only the mandatory IEs for each message type  are  added  to
the  message  structure  by  the qcc_create call.  The addi-
tional parameters to the qcc_create functions allow the user
to  define most of the information contained in those manda-
tory IEs; however, in some cases default values are assumed.
Those  values, as well as the additional parameters for each
function, are indicated in the following paragraphs describ-
ing each function call.

qcc_create_setup() creates a setup  message  structure  con-
taining  the  following  Information  Elements:  ATM traffic
descriptor (called ATM cell  rate  in  UNI  3.0),  broadband
bearer  capability, called party number, and quality of ser-
vice parameter. The user must pass in  the  destination  ATM
address for the called party number IE (atm_address_t format
is defined in the <atm/types.h> header file).  The following
default  values  are used for the remaining Information Ele-
ments:

     ATM Traffic Descriptor:
          best effort; line rate is used for the forward and
          backward peak rates

     Broadband Bearer Capability:
          Bearer Class X, no indication for traffic type and
          timing  requirements, not susceptible to clipping,
          and point-to-point user plane

     Called Party Number:
          ATM Endsystem (NSAP) address type

     Quality of Service:
          Forward and backward class unspecified

qcc_create_alerting() creates the structure for an  alerting
message, which is supported only under UNI 4.0. The alerting
message contains no mandatory IEs; only the  message  header
is filled in.

qcc_create_call_proceeding() creates  the  structure  for  a
```

CODE EXAMPLE 2-3 qcc_create(3) Man Page *(Continued)*

```
     call  proceeding  message,  which contains no mandatory IEs.
     Only the message header is filled in.

     qcc_create_connect() creates the  structure  for  a  connect
     message,  which  also contains no mandatory IEs. Again, only
     the required header is filled in.   The  same  is  true  for
     qcc_create_connect_ack.

     qcc_create_release() creates  a  release  message  structure
     containing  a  cause  IE,  for which the user must pass in a
     cause value.  The  possible  values  can  be  found  in  the
     <atm/qccdefs.h>  header  file.  By default, no diagnostic is
     included and the user location is assigned.

     qcc_create_release_complete() creates the  structure  for  a
     release  complete  message, which contains no mandatory IEs.
     Only the message header is filled in.

     qcc_create_status_enquiry() creates a status enquiry message
     structure, which contains no mandatory IEs. Only the message
     header is filled in.

     qcc_create_status() builds a status message structure,  con-
     taining  two  mandatory  IEs: call state and cause. The user
     should pass in value for both the callstate and  the  cause;
     possible  values  may be found in the <atm/qccdefs.h> header
     file. In the cause IE, no diagnostic  is  included  and  the
     user location is assigned.

     qcc_create_notify() builds a notify message structure, which
     is only supported under UNI 4.0. The message contains a sin-
     gle mandatory IE, the notification indicator, which contains
     a  buffer  of  user-specified  data. The maximum size of the
     buffer is 16 bytes, defined as  QCC_MAX_NOTIFICATION_LEN  in
     <atm/qcc.h>.  The  user should allocate a buffer and pass in
     the buffer length, contentlen, and a pointer to the  buffer,
     contentp.

     qcc_create_restart() creates a  restart  message  structure,
     containing  the  mandatory restart indicator IE, and option-
     ally the connection identifier IE.  The user should pass  in
     a     value     for     the     restart     indicator,     either
     RESTART_INDICATED_VC or RESTART_ALL_VCS. If a  non-zero  vci
     parameter is passed in, the connection identifier IE is also
     included in the message, using a default vpci of 0  and  the
     vci parameter value.
```

CODE EXAMPLE 2-3 `qcc_create(3)` Man Page *(Continued)*

```
     qcc_create_add_party()  constructs  an  add  party   message
     structure.   It  includes  the mandatory called party number
     and endpoint reference IEs.   The  user  should  pass  in  a
     pointer  to  the  called  number  and  an endpoint reference
     value; for the called party  number,  ATM  Endsystem  (NSAP)
     address type is assumed.

     qcc_create_add_party_ack() fills in an add party ack message
     structure  with  the  endpoint  reference  IE. The endpt_ref
     parameter value is used.

     qcc_create_party_alerting() creates a party alerting message
     structure  with  the  endpoint  reference IE, which uses the
     endpt_ref parameter.  This message type  is  only  supported
     under UNI 4.0.

     qcc_create_add_party_reject() fills the cause  and  endpoint
     reference  IEs  into an add party reject structure. The user
     should provide the cause and endpoint reference value;  pos-
     sible cause values are defined in the <atm/qccdefs.h> header
     file. By default, no diagnostic is  included  and  the  user
     location is assigned in the cause IE.

     qcc_create_drop_party() fills the cause and endpoint  refer-
     ence  IEs  into a drop party structure. The user should pass
     in the cause and endpoint reference values;  possible  cause
     values  are  defined  in the <atm/qccdefs.h> header file. By
     default, no diagnostic is included and the user location  is
     assigned in the cause IE.

     qcc_create_drop_party_ack() fills in only the mandatory end-
     point reference IE, requiring the endpt_ref parameter.

     qcc_create_leaf_setup_fail() creates a leaf setup fail  mes-
     sage  structure,  with  three  mandatory  IEs.  The cause IE
     requires the cause parameter, which should  be  one  of  the
     cause  values  defined in <atm/qccdefs.h>; the called number
     IE requires the destination ATM address, dst_addrp; and  the
     leaf  number  IE requires the leaf_num parameter.  This mes-
     sage is only supported under UNI 4.0.

     qcc_create_leaf_setup_req() creates  a  leaf  setup  request
     message structure, with four mandatory IEs. Both the calling
     party and called party number IEs are  required,  using  the
     source  and  destination  ATM  addresses,  passed  in in the
```

CODE EXAMPLE 2-3 `qcc_create(3)` Man Page *(Continued)*

```
     src_addrp and dst_addrp parameters, respectively.  The  leaf
     initiated  join  call  identifier IE requires the lij_callid
     parameter. The final required IE, the  leaf  number  IE,  is
     inserted  as  a  placeholder; the actual leaf number will be
     assigned and filled in by  the  q93b  driver.   It  will  be
     returned  in  the  callref  field  of  the  qcc  header of a
     leaf_setup_ack  message,  much  as  the  call  reference  is
     returned in a setup_ack message in the setup case.  Refer to
     the description of the qcc_bld_leaf_setup_req() function for
     more  details  on  this  process.  This message is only sup-
     ported under UNI 4.0.

RETURN VALUES
     All functions return 0 on success and -1 on error.

EXAMPLES
     The following code fragment creates a setup message, adds an
     optional  AAL  Parameters IE, packs the message into streams
     buffers, and sends it downstream.

          #include <atm/limits.h>
          #include <atm/qcc.h>
          #include <atm/qcctypes.h>

          char    ifname[QCC_MAX_IFNAME_LEN] = "ba0";
          int     calltag = 0x1234;
          int     forward_sdusize = 0x2378;
          int     backward_sdusize = 0x2378;
          qcc_msg_t          msgstruct;
          qcc_setup_t        setup;
          qcc_ie_t           iestruct;
          qcc_aal_params_t   aal;
          struct strbuf      ctl, data;
          char               ctlbuf[QCC_MAX_CTL_LEN];
          char               databuf[QCC_MAX_DATA_LEN];

          atm_addr_t     dst_addr = {
               0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
               0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
               0x08, 0x00, 0x20, 0x1a, 0xb6, 0xb9, 0x00
          };

          ctl.buf = ctlbuf;
          data.buf = databuf;
          ctl.maxlen = QCC_MAX_CTL_LEN;
          data.maxlen = QCC_MAX_DATA_LEN;
```

CODE EXAMPLE 2-3 qcc_create(3) Man Page *(Continued)*

```
          if ((qcc_create_setup(&setup, ifname,
                                calltag, dst_addr)) < 0) {
               printf("qcc_create_setup failed\n");
               exit (-1);
          }

          msgstruct.type = QCC_SETUP;
          msgstruct.msg.setup = &setup;

          aal.type = AAL_TYPE_5;
          aal.info.aal5.forward_max = forward_sdusize;
          aal.info.aal5.backward_max = backward_sdusize;
          aal.info.aal5.mode = MESSAGE_MODE;
          aal.info.aal5.sscs_type = SSCS_TYPE_NULL;

          iestruct.type = QCC_AAL_PARAMETERS;
          iestruct.ie.aal_params = &aal;

          if ((qcc_set_ie(&msgstruct, &iestruct)) < 0) {
               printf("qcc_set_ie failed\n");
               exit (-1);
          }

          if ((qcc_pack_setup(&ctl, &data,
                              msgstruct.msg.setup)) < 0) {
               printf("qcc_pack_setup failed\n");
               exit (-1);
          }

          if (putmsg(fd, &ctl, &data, 0) < 0) {
               perror("putmsg");
               exit (-1);
          }

SEE ALSO
     qcc_set_ie(3),  qcc_pack(3),  qcc_unpack(3),   qcc_parse(3),
     qcc_util(3), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
```

CODE EXAMPLE 2-3 `qcc_create(3)` Man Page *(Continued)*

```
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.  These message types, if sent on an interface
     configured for UNI 3.0 or 3.1, will be discarded by the q93b
     driver  and  will  not  be sent out to the network.  The UNI
     4.0-specific messages are Alerting, Notify, Party  Alerting,
     Leaf  Setup Fail, and Leaf Setup Request, and are identified
     in the applicable function descriptions.
```

qcc_len(3)

CODE EXAMPLE 2-4 qcc_len(3) Man Page

```
qcc_len(3)             C Library Functions             qcc_len(3)



NAME
     qcc_len,  qcc_bld_setup_datalen,   qcc_bld_alerting_datalen,
     qcc_bld_call_proceeding_datalen,    qcc_bld_connect_datalen,
     qcc_bld_connect_ack_datalen,        qcc_bld_release_datalen,
     qcc_bld_release_complete_datalen,
     qcc_bld_status_enquiry_datalen,      qcc_bld_notify_datalen,
     qcc_bld_status_datalen,             qcc_bld_restart_datalen,
     qcc_bld_restart_ack_datalen,      qcc_bld_add_party_datalen,
     qcc_bld_add_party_ack_datalen,
     qcc_bld_party_alerting_datalen,
     qcc_bld_add_party_reject_datalen,
     qcc_bld_drop_party_datalen,  qcc_bld_drop_party_ack_datalen,
     qcc_bld_leaf_setup_fail_datalen,
     qcc_bld_leaf_setup_req_datalen,         qcc_max_bld_datalen,
     qcc_ctl_len - get length of Q.2931 messages

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/qcc.h> #include <atm/limits.h>

     size_t qcc_bld_setup_datalen();

     size_t qcc_bld_alerting_datalen();

     size_t qcc_bld_call_proceeding_datalen();

     size_t qcc_bld_connect_datalen();

     size_t qcc_bld_connect_ack_datalen();

     size_t qcc_bld_release_datalen();

     size_t qcc_bld_release_complete_datalen();

     size_t qcc_bld_status_enquiry_datalen();
```

CODE EXAMPLE 2-4 qcc_len(3) Man Page *(Continued)*

```
     size_t qcc_bld_notify_datalen();

     size_t qcc_bld_status_datalen();

     size_t qcc_bld_restart_datalen();

     size_t qcc_bld_restart_ack_datalen();

     size_t qcc_bld_add_party_datalen();

     size_t qcc_bld_add_party_ack_datalen();

     size_t qcc_bld_party_alerting_datalen();

     size_t qcc_bld_add_party_reject_datalen();

     size_t qcc_bld_drop_party_datalen();

     size_t qcc_bld_drop_party_ack_datalen();

     size_t qcc_bld_leaf_setup_fail_datalen();

     size_t qcc_bld_leaf_setup_req_datalen();

     size_t qcc_max_bld_datalen();

     size_t qcc_ctl_len();

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions may be used to determine appropriate  buffer
     sizes  for the control and data buffers that are passed into
     qcc_bld(3)   functions.   For   the   data   buffer,   the
     qcc_bld_*_datalen()  functions  will return the maximum size
     of a particular message type.  qcc_max_bld_datalen() returns
     the  maximum  size  of  all  Q.2931 message types.  A buffer
     allocated for this size will be able  to  hold  any  message
```

CODE EXAMPLE 2-4 qcc_len(3) Man Page *(Continued)*

```
     type.  For the control buffer, qcc_ctl_len() will return the
     required size.

SEE ALSO
     qcc_bld(3), qcc_parse(3), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message types will be ignored by the
     q93b driver if used on an interface which is configured  for
     UNI 3.0 or 3.1.  The UNI 4.0-specific messages are Alerting,
     Notify, Party Alerting, Leaf  Setup  Fail,  and  Leaf  Setup
     Request.
```

qcc_pack(3)

CODE EXAMPLE 2-5 qcc_pack(3) Man Page

```
qcc_pack(3)            C Library Functions            qcc_pack(3)



NAME
     qcc_pack,         qcc_pack_setup,         qcc_pack_alerting,
     qcc_pack_call_proceeding,                  qcc_pack_connect,
     qcc_pack_connect_ack,                      qcc_pack_release,
     qcc_pack_release_complete,                  qcc_pack_status,
     qcc_pack_status_enq,   qcc_pack_notify,    qcc_pack_restart,
     qcc_pack_restart_ack,                    qcc_pack_add_party,
     qcc_pack_add_party_ack,             qcc_pack_party_alerting,
     qcc_pack_add_party_reject,              qcc_pack_drop_party,
     qcc_pack_drop_party_ack,           qcc_pack_leaf_setup_fail,
     qcc_pack_leaf_setup_req  -  encode  Q.2931 message structure
     information and pack into streams buffers

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/types.h>
     #include <atm/qcc.h>

     int qcc_pack_setup(strbuf_t *ctlp, strbuf_t *datap,
           qcc_setup_t *msgp);

     int qcc_pack_alerting(strbuf_t *ctlp, strbuf_t *datap,
           qcc_alerting_t *msgp);

     int qcc_pack_call_proceeding(strbuf_t *ctlp,
           strbuf_t *datap, qcc_call_proc_t *msgp);

     int qcc_pack_connect(strbuf_t *ctlp, strbuf_t *datap,
           qcc_connect_t *msgp);

     int qcc_pack_connect_ack(strbuf_t *ctlp, strbuf_t *datap,
           qcc_connect_ack_t *msgp);

     int qcc_pack_release(strbuf_t *ctlp, strbuf_t *datap,
           qcc_release_t *msgp);
```

CODE EXAMPLE 2-5 `qcc_pack(3)` Man Page *(Continued)*

```
     int qcc_pack_release_complete(strbuf_t *ctlp,
           strbuf_t *datap, qcc_release_complete_t *msgp);

     int qcc_pack_status_enq(strbuf_t *ctlp, strbuf_t *datap,
           qcc_status_enq_t *msgp);

     int qcc_pack_status(strbuf_t *ctlp, strbuf_t *datap,
           qcc_status_t *msgp);

     int qcc_pack_notify(strbuf_t *ctlp, strbuf_t *datap,
           qcc_notify_t *msgp);

     int qcc_pack_restart(strbuf_t *ctlp, strbuf_t *datap,
           qcc_restart_t *msgp);

     int qcc_pack_restart_ack(strbuf_t *ctlp, strbuf_t *datap,
           qcc_restart_ack_t *msgp);

     int qcc_pack_add_party(strbuf_t *ctlp, strbuf_t *datap,
           qcc_add_party_t *msgp);

     int qcc_pack_add_party_ack(strbuf_t *ctlp, strbuf_t *datap,
           qcc_add_party_ack_t *msgp);

     int qcc_pack_party_alerting(strbuf_t *ctlp, strbuf_t *datap,
           qcc_party_alerting_t *msgp);

     int qcc_pack_add_party_reject(strbuf_t *ctlp,
           strbuf_t *datap, qcc_add_party_reject_t *msgp);

     int qcc_pack_drop_party(strbuf_t *ctlp, strbuf_t *datap,
           qcc_drop_party_t *msgp);

     int qcc_pack_drop_party_ack(strbuf_t *ctlp, strbuf_t *datap,
           qcc_drop_party_ack_t *msgp);

     int qcc_pack_leaf_setup_fail(strbuf_t *ctlp,
           strbuf_t *datap, qcc_leaf_setup_fail_t *msgp);

     int qcc_pack_leaf_setup_req(strbuf_t *ctlp, strbuf_t *datap,
           qcc_leaf_setup_req_t *msgp);

MT-LEVEL
     Safe.
```

CODE EXAMPLE 2-5 `qcc_pack(3)` Man Page *(Continued)*

```
AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions take message structures as input and  encode
     the  information  contained  in  the  structure  to create a
     Q.2931 message, which is then  packed  into  streams  buffer
     structures.  The Q.2931 protocol is used for ATM signalling;
     a full description of the message  format  and  use  can  be
     found  in  the ATM Forum's User Network Interface Specifica-
     tion, V3.0, V3.1, or V4.0. The encoded messages will conform
     to  the version of the UNI Specification which is configured
     on the indicated interface. The functions  may  be  used  by
     processes which are running in user space.

     Message structures should be filled using the  qcc_create(3)
     and  qcc_set_ie(3)  functions  before calling qcc_pack func-
     tions.

     In general, no error checking is performed on the data  that
     is  passed  in.   Whatever  data is contained in the message
     structure will be placed  in  the  encoded  message  without
     examination.

     Each function requires 3 parameters: ctlp and  datap,  which
     are  pointers  to  strbuf_t  buffers;  and  msgp, which is a
     pointer to the appropriate message structure.

     ctlp and datap make up the control and data portions of  the
     constructed message, corresponding to the M_PROTO and M_DATA
     blocks of the message that will be passed  downstream.   The
     buffer  fields  in the structures which ctlp and datap point
     to (ctlp->buf and datap->buf) must be allocated before  cal-
     ling a qcc_pack_* function; size information may be obtained
     using the qcc_bld_*_datalen()  functions  (see  qcc_len(3)).
     After successful return from a qcc_pack_* function, the mes-
     sage may be passed down an open stream using  the  putmsg(2)
     function,  with  ctlp and datap as the buffer parameters for
     putmsg.

RETURN VALUES
     All functions return 0 on success and -1 on error.
```

CODE EXAMPLE 2-5 `qcc_pack(3)` Man Page *(Continued)*

```
EXAMPLES
     For an example using qcc_pack_setup, see the example in  the
     qcc_create(3) man page.

SEE ALSO
     qcc_len(3),   qcc_create(3),   qcc_set_ie(3),   qcc_util(3),
     q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message types will be ignored by the
     q93b driver if used on an interface which is configured  for
     UNI 3.0 or 3.1.  The UNI 4.0-specific messages are Alerting,
     Notify, Party Alerting, Leaf  Setup  Fail,  and  Leaf  Setup
     Request.
```

qcc_parse(3)

CODE EXAMPLE 2-6 qcc_parse(3) Man Page

```
qcc_parse(3)           C Library Functions           qcc_parse(3)



NAME
     qcc_parse,       qcc_parse_setup,        qcc_parse_alerting,
     qcc_parse_call_proceeding,                qcc_parse_connect,
     qcc_parse_release,               qcc_parse_release_complete,
     qcc_parse_status_enquiry,                  qcc_parse_notify,
     qcc_parse_status, qcc_parse_restart,  qcc_parse_restart_ack,
     qcc_parse_add_party,                qcc_parse_add_party_ack,
     qcc_parse_party_alerting,        qcc_parse_add_party_reject,
     qcc_parse_drop_party,              qcc_parse_drop_party_ack,
     qcc_parse_leaf_setup_fail,         qcc_parse_leaf_setup_req,
     qcc_get_hdr - parse Q.2931 messages

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/types.h>
     #include <atm/qcc.h>

     int qcc_parse_setup(strbuf_t *datap, int *vcip,
           int *forward_sdusizep, int *backward_sdusizep,
           atm_addr_t *src_addrp, atm_addr_t *dst_addrp,
           int *sapp, int *endpt_refp);

     int qcc_parse_alerting(strbuf_t *datap, int *vcip,
           int *endpt_refp);

     int qcc_parse_call_proceeding(strbuf_t *datap, int *vcip,
           int *endpt_refp);

     int qcc_parse_connect(strbuf_t *datap, int *vcip,
           int *forward_sdusizep, int *backward_sdusizep,
           int *endpt_refp);

     int qcc_parse_release(strbuf_t *datap, int *causep);

     int qcc_parse_release_complete(strbuf_t *datap,
```

CODE EXAMPLE 2-6 `qcc_parse(3)` Man Page *(Continued)*

```
      int *causep);

int qcc_parse_status_enquiry(strbuf_t *datap,
      int *endpt_refp);

int qcc_parse_notify(strbuf_t *datap, int *contentlenp,
      u_char *contentp, int *endpt_refp);

int qcc_parse_status(strbuf_t *datap, int *callstatep,
      int *causep, int *endpt_refp, int *endpt_statep);

int qcc_parse_restart(strbuf_t *datap, int *vcip,
      int *rstallp);

int qcc_parse_restart_ack(strbuf_t *datap, int *vcip,
      int *rstallp);

int qcc_parse_add_party(strbuf_t *datap,
      int *forward_sdusize, int *backward_sdusize,
      atm_address_t *src_addrp, atm_address_t *dst_addrp,
      int *sapp, int *endpt_refp);

int qcc_parse_add_party_ack(strbuf_t *datap,
      int *endpt_refp);

int qcc_parse_party_alerting(strbuf_t *datap,
      int *endpt_refp);

int qcc_parse_add_party_reject(strbuf_t *datap, int *causep,
      int *endpt_refp);

int qcc_parse_drop_party(strbuf_t *datap, int *causep,
      int *endpt_refp);

int qcc_parse_drop_party_ack(strbuf_t *datap, int *causep,
      int *endpt_refp);

int qcc_parse_leaf_setup_fail(strbuf_t *datap, int *causep,
      atm_address_t *dst_addrp, int *leaf_nump);

int qcc_parse_leaf_setup_req(strbuf_t *datap,
      atm_address_t *src_addrp, atm_address_t *dst_addrp,
      int *lij_callidp, int *leaf_nump);

qcc_hdr_t *qcc_get_hdr(strbuf_t *ctlp);
```

CODE EXAMPLE 2-6 qcc_parse(3) Man Page *(Continued)*

```
MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions parse the various messages that make up  the
     Q.2931  protocol  which  is used for ATM signalling.  A full
     description of the message format and use can  be  found  in
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0.  Messages conforming to both versions will be
     parsed.   The  functions  may be used by processes which are
     running in user space.

     Each function requires a  minimum  of  1  parameter:  datap,
     which  is  a pointer to a strbuf_t buffer, or in the case of
     qcc_get_hdr, ctlp, which is also a  pointer  to  a  strbuf_t
     buffer.

     datap is the data portion of a STREAMS message,  correspond-
     ing to the M_DATA block of the message that is received from
     downstream. After receiving a message  using  the  getmsg(2)
     function,  the message type may be examined and an appropri-
     ate parsing routing called to extract information  from  the
     signalling message.

     ctlp  is  the  control  portion  of   a   STREAMS   message,
     corresponding  to  the  M_PROTO block of the message that is
     received from downstream. After receiving  a  message  using
     the  getmsg(2)  function, qcc_get_hdr may be used to extract
     the Q.2931 header structure from the control buffer received
     from  getmsg(2).  The  Q.2931  header  type,  qcc_hdr_t,  is
     defined in <atm/types.h>.

     Other parameters for each function depend  on  the  type  of
     information  that is available in each message type.  In all
     cases, certain IEs are examined in each  message,  as  indi-
     cated  below.  If those IEs exist, the data that is expected
     from them is retrieved, but no error message is sent if they
     do  not  exist;  the value of the parameter is set to -1 for
     any data that was expected from that particular  IE.   Also,
     IEs that are not expected are ignored. If the user wishes to
```

CODE EXAMPLE 2-6 `qcc_parse(3)` Man Page *(Continued)*

```
     ignore any of the parameters of a parse function, passing in
     a  NULL  pointer for that parameter is allowed so that space
     need not be allocated for the unnecessary parameter.

     qcc_parse_setup() parses a setup message containing the fol-
     lowing  Information  Elements: AAL parameters, ATM user cell
     rate, broadband bearer capability, called party number, cal-
     ling  party number, quality of service parameter, connection
     identifier, broadband higher layer information, and endpoint
     reference.  The  endpoint  reference  IE is only included in
     setup messages for point-to-multipoint calls, The  following
     table  matches  the  data that is retrieved from the message
     with the IE from which it is parsed.

            DATA RETRIEVED              INFORMATION ELEMENT
            vci                         connection identifier
            forward sdusize             AAL parameters
            backward sdusize            AAL parameters
            source address              calling party number
            destination address         called party number
            sap                         broadband higher layer
            endpoint reference id       endpoint reference

     qcc_parse_alerting() parses an alerting message. The  alert-
     ing  message  is new in UNI 4.0; if received on an interface
     configured for uni 3.0 or 3.1, it will  be  dropped  by  the
     q93b driver.  The IEs examined by this function are the con-
     nection identifier IE, from which the vci is parsed, and the
     endpoint reference IE, from which the endpt_ref parameter is
     parsed.  The endpoint  reference  IE  is  only  included  in
     alerting messages for point-to-multipoint calls.

     qcc_parse_call_proceeding() parses a call proceeding message
     containing  a connection identifier IE, which is used to set
     the value of vci, and an endpoint reference IE, setting  the
     value  of  endpt_ref.  The  endpoint  reference  IE  is only
     included in call proceeding messages for point-to-multipoint
     calls.

     qcc_parse_connect() parses a connect message  containing  an
     AAL  parameters IE, setting the forward and backward sdusize
     values, a connection identifier IE,  setting  the  value  of
     vci,  and  an  endpoint  reference  IE, setting the value of
     endpt_ref. The endpoint reference IE  is  only  included  in
     connect messages for point-to-multipoint calls.
```

CODE EXAMPLE 2-6 qcc_parse(3) Man Page *(Continued)*

```
     qcc_parse_release() parses a cause  IE,  setting  the  cause
     value.  A listing of the possible values can be found in the
     <atm/qcc.h>   header   file.   The   same   is   true    for
     qcc_parse_release_complete.

     qcc_parse_status_enquiry() parses a status  enquiry  message
     containing  an  endpoint  reference IE, setting the value of
     endpt_ref.  The endpoint reference IE is only included  when
     enquiring about a party state in a point-to-multipoint call.

     qcc_parse_status() parses a status message.   The  IEs  that
     are  parsed  are  call state, cause, endpoint reference, and
     endpoint state.  The call state and cause IEs  are  used  to
     set  the value of the parameters callstate and cause; possi-
     ble  values  for  both  parameters  may  be  found  in   the
     <atm/qcc.h> header file. The endpoint reference and endpoint
     state IEs will be used to set the values  of  the  endpt_ref
     and  endpt_state parameters; they are included if an enquiry
     is made about a party state in a point-to-multipoint call or
     to report an error condition in a point-to-multipoint call.

     qcc_parse_notify() parses a notify message,  which  is  only
     supported under UNI 4.0. The notification indicator and end-
     point reference IEs are parsed; from the notification  indi-
     cator,  the  contentlenp  and contentp parameters are filled
     in, with the maximum buffer size copied being 16 bytes.   If
     the  size  contained in the message is greater than 16 bytes
     (QCC_MAX_NOTIFICATION_LEN,  defined  in  <atm/qcc.h>),   the
     first 16 bytes are copied, contentlenp is set to contain the
     copied length of 16 bytes, and the  overflow  flag  is  set.
     From  the  endpoint  reference  IE, endpt_refp is filled in.
     The endpoint reference  IE  is  only  present  on  point-to-
     multipoint calls.

     qcc_parse_restart() parses a restart message containing  two
     possible  IEs:  connection identifier and restart indicator.
     The restart indicator IE is used to set the value of rstall;
     this  parameter  indicates  whether  a particular vci or all
     vcis are to be restarted  (rstall  =  1  implies  all  vcis,
     rstall  = 0 implies a particular vci).  The connection iden-
     tifier identifies the particular vci.   In  this  case,  the
     value  of  the parameter vci is set to 0 if there is no con-
     nection identifier IE  in  the  message.   The  same  format
     applies to the qcc_parse_restart_ack() function.

     qcc_parse_add_party() parses an add party message containing
```

CODE EXAMPLE 2-6 `qcc_parse(3)` Man Page *(Continued)*

```
      several  possible  IEs. They include AAL parameters, calling
      party number, called party number,  broadband  higher  layer
      information,  and  endpoint  reference.  The following table
      matches the data that is retrieved from the message with the
      IE from which it is parsed.

             DATA RETRIEVED              INFORMATION ELEMENT
             forward sdusize             AAL parameters
             backward sdusize            AAL parameters
             source address              calling party number
             destination address         called party number
             sap                         broadband higher layer
             endpoint reference id       endpoint reference

      qcc_parse_add_party_ack()  extracts  an  endpoint  reference
      value  from  the  endpoint  reference IE in an add party ack
      message.

      qcc_parse_party_alerting() extracts  an  endpoint  reference
      value  from  the  endpoint  reference IE in a party alerting
      message.  This message is specific to UNI 4.0.

      qcc_parse_add_party_reject() parses an add party reject mes-
      sage  possibly containing a cause IE, from which it extracts
      the cause value, and an endpoint reference IE, from which it
      extracts the endpoint reference value. Possible cause values
      may be found in the header file <atm/qcc.h>.

      qcc_parse_drop_party() extracts an endpoint reference  value
      and  a cause value from those respective IEs in a drop party
      message. The same is true for qcc_parse_drop_party_ack().

      qcc_parse_leaf_setup_fail() extracts a cause value  (defined
      in  <atm/qcc.h>)  from  the  cause IE; a destination address
      from the called number IE; and a leaf number from  the  leaf
      number  IE.  The  leaf setup fail message is specific to UNI
      4.0.

      qcc_parse_leaf_setup_req() parses a leaf setup request  mes-
      sage,  which is specific to UNI 4.0.  The calling number and
      called number IEs are parsed, yielding the source and desti-
      nation  ATM  addresses,  respectively; in addition, the leaf
      initiated join call identifier IE is parsed  to  obtain  the
      leaf initiated join callid, and the leaf number IE is parsed
      for the leaf number.
```

CODE EXAMPLE 2-6 `qcc_parse(3)` Man Page *(Continued)*

```
     qcc_get_hdr() extracts the Q.2931 header  from  the  control
     buffer  received  in  getmsg(2).  A  pointer to this buffer,
     ctlp, is passed in to the function, and  a  pointer  to  the
     header of type qcc_hdr_t is returned on success. On failure,
     a null pointer is returned.

RETURN VALUES
     All functions, with the exception of qcc_get_hdr,  return  0
     on   success  and  -1  on  error.   The  return  values  for
     qcc_get_hdr are described above.

EXAMPLES
     The following code fragment receives and parses a setup mes-
     sage.

          #include <atm/types.h>
          #include <atm/qcc.h>
          #include <atm/limits.h>

          void
          wait_for_setup(int fd);
          {
               int             vci;
               int             forward_sdusize;
               int             backward_sdusize;
               int             sap;
               int             flags = 0;
               atm_addr_t      src_addr;
               atm_addr_t      dst_addr;
               qcc_hdr_t       *hdrp;
               struct strbuf   ctl, data;
               char            ctlbuf[QCC_MAX_CTL_LEN];
               char            databuf[QCC_MAX_DATA_LEN];

               ctl.buf = ctlbuf;
               data.buf = databuf;
               ctl.len = data.len = 0;
               ctl.maxlen = QCC_MAX_CTL_LEN;
               data.maxlen = QCC_MAX_DATA_LEN;

               if (getmsg(fd, &ctl, &data, &flags) < 0) {
                    perror("getmsg");
                    exit (-1);
               }

               hdrp = qcc_get_hdr(&ctl);
```

CODE EXAMPLE 2-6 `qcc_parse(3)` Man Page *(Continued)*

```
               if ((hdrp) && (hdrp->type == QCC_SETUP)) {
                    if ((qcc_parse_setup(&data, &vci, &forward_sdusize,
                               &backward_sdusize, &src_addr,
                               &dst_addr, &sap, NULL)) < 0) {
                         printf("parse_setup failed\n");
                         exit (-1);
                    }
                    printf("parse_setup: vci = 0x%x, sap = 0x%x\n",
                               vci, sap);
               }
          }

SEE ALSO
     qcc_bld(3), qcc_len(3), qcc_util(3), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message  types,  if  received  on an
     interface configured for UNI 3.0 or 3.1, will  be  discarded
     by  the  q93b  driver  and  will  not be sent up to the user
     applications.  The UNI 4.0-specific messages  are  Alerting,
     Notify, Party Alerting, Leaf Setup Fail, and Leaf Setup Req,
     and are identified in the applicable function descriptions.
```

qcc_set_ie(3)

CODE EXAMPLE 2-7 qcc_set_ie(3) Man Page

```
qcc_set_ie(3)          C Library Functions          qcc_set_ie(3)



NAME
     qcc_set_ie - add or update Information Elements in a  Q.2931
     message structure

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/qcc.h>
     #include <atm/qcctypes.h>

     int qcc_set_ie(qcc_msg_t *msgp, qcc_ie_t *iep);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     This function adds a new or changes an existing  Information
     Element in Q.2931 messages.  The Q.2931 protocol is used for
     ATM signalling.  A full description of  the  message  format
     and  use can be found in the ATM Forum's User Network Inter-
     face Specification, V3.0 or V3.1. The function may  be  used
     by processes which are running in user space.

     A message  structure  should  first  be  created  using  the
     appropriate  qcc_create(3)  function  call.  IEs may then be
     added or changed using qcc_set_ie.  When the message  struc-
     ture   has  been  completely  specified,  the  corresponding
     qcc_pack(3) function should be called to translate the  mes-
     sage structure into the correct encoded format, contained in
     streams buffers which may be passed to the  putmsg(2)  func-
```

CODE EXAMPLE 2-7 qcc_set_ie(3) Man Page *(Continued)*

```
     tion.

     In general, no error checking is performed on the data  that
     is  passed in.  Whatever data is passed in will be placed in
     the message that is built  without  examination.   The  user
     should  insure that the values passed in in the IE structure
     conform with the UNI version (3.0 or 3.1) that is running.

     The function requires 2 parameters: msgp, which is a pointer
     to  the  appropriate  message structure; and iep, which is a
     pointer to the new IE structure.  The message and IE  struc-
     ture types are defined in the <atm/qcctypes.h> header file.

     The structure to which msgp points must be allocated by  the
     calling  user.   The structure pointed to by iep should have
     the desired values filled in to its fields, and the  "valid"
     field should be set to 1.  A value of 0 in the "valid" field
     indicates that the IE should not be included in the message.

     The fields of each Information Element structure  and  their
     interpretations  are  described in the following paragraphs.
     Possible  values  for  IE  fields   are   defined   in   the
     <atm/qccdefs.h> header file.

     qcc_aal_params_t
          Currently, the only ATM Adaptation Layer  supported  on
          SunATM products is AAL 5.  However, to allow for future
          changes, the aal parameters ie type consists of a field
          identifying  the aal and a union of structures for each
          aal, called "info."  The aal  5  structure  contains  4
          fields: forward_max and backward_max for the SDU sizes,
          mode, and sscs_type. The sscs_type is only valid in UNI
          3.0;  therefore,  a  value of 0 for sscs_type indicates
          that that field should not be included.

     qcc_traffic_desc_t
          The ATM Traffic Descriptor IE (called User Cell Rate in
          UNI  3.0)  contains  a  large  set of traffic parameter
          values.  Two parameters  do  not  have  numeric  values
          associated;  they  are either included or not.  The are
          represented by two  fields,  best_effort  and  tagging,
          that  are  either  set  to  1 if the parameter is to be
          included or set to 0  if  it  is  not.   The  remaining
          parameters  all  have  numeric  values  associated with
          them.  Since 0 is a valid value for  these  parameters,
          an  additional  field,  params,  is  included in the IE
```

CODE EXAMPLE 2-7 qcc_set_ie(3) Man Page *(Continued)*

```
         structure which indicates  which  of  these  should  be
         included   in   the  message.   Each  parameter  has  a
         corresponding bit in the params field, which, when set,
         indicates  that the parameter should be included. Flags
         are defined  for  this  field  in  the  <atm/qccdefs.h>
         header file.

    qcc_bbc_t
         The Broadband Bearer Capability  IE  fields  correspond
         directly to the options for this IE.  The fields are:

             class           Bearer Class
             type            Traffic Type
             timing          Timing Requirements
             clipping        Susceptibility to Clipping
             userplane       User plane connection configuration

    qcc_bhli_t
         The Broadband High Layer Information IE structure  con-
         tains 3 fields which specify the IE contents.  They are
         type, which identifies the High Layer Information Type;
         infolen,  which  indicates the number of octets of high
         layer information is to be included in the message (the
         maximum  is  8  octets),  and finally an array of bytes
         called info  which  contains  the  information  octets,
         called  info.  The octets should be placed in the first
         infolen elements of the array.

    qcc_blli_t
         The Broadband  Low  Layer  Information  IE  contains  2
         fields  to  specify the IE contents.  The first, layer,
         is an integer which specifies which layer  protocol  is
         being  specified,  layer  1,  2, or 3.  The second is a
         union, with unique structures for layer 2 and layer  3.
         For  both  layer  2 and layer 3 IEs, the protocol value
         will be examined and the correct coding format will  be
         used for that protocol.  Therefore, only the applicable
         fields from the layer structure will be  used  for  the
         specified protocol type.

         Layer 2 fields:
             protocol     User information layer 2 protocol
             mode         Mode of operation
             windowsize   Window size (k)
             userspec     User specified layer 2 protocol
                          information
```

CODE EXAMPLE 2-7 `qcc_set_ie(3)` Man Page *(Continued)*

```
     Layer 3 fields:
         protocol     User information layer 3 protocol
         mode         Mode of operation
         pktsize      Default packet size
         windowsize   Packet window size
         userspec     User specified layer 3 protocol
                      information
         ipi          8-bit Initial Protocol Identifier for
                      ISO/IEC TR 9577
         oui          24-bit organization unique identifier
                      for ISO/IEC TR 9577 and IEEE 802.1 SNAP
         pid          16-bit protocol identifier for ISO/IEC
                      TR 9577 and IEEE 802.1 SNAP


qcc_call_state_t
     There is only one informational field in the Call State
     IE structure:  state, specifying the call state.

qcc_called_num_t
     The Called Party Number IE structure contains a  planid
     field,  which  specifies  the Addressing/Numbering Plan
     Identification.  The Type of Number is  based  on  this
     value  as  well.   There  is  also an address field, to
     specify a 20-byte address.

qcc_called_subaddr_t
     The Called Party Subaddress  IE  structure  contains  a
     type field, which specifies the Type of Subaddress, and
     a 20-byte address field.

qcc_calling_num_t
     In addition to the 20-byte address field,  the  Calling
     Party  Number  IE  structure contains several fields to
     describe the intended interpretation  of  the  address.
     They are:

         planid         Addressing/Numbering Plan
                        Identification
         presentation   Presentation indicator
         screening      Screening indicator

qcc_calling_subaddr_t
     The structure for the Calling Party  Subaddress  IE  is
```

CODE EXAMPLE 2-7 qcc_set_ie(3) Man Page *(Continued)*

```
          identical to that of the Called Party Subaddress IE.

     qcc_cause_t
          The Cause IE structure contains a location field and  a
          cause  field.   In addition, it contains an array of 28
          octets, diag, for diagnostic information. The number of
          diagnostic  octets  included  in  the  array  should be
          specified in the diaglen field.

     qcc_conn_id_t
          The Connection Identifier IE structure contains a  vpci
          and  a  vci  field.  Note  that  currently,  the SunATM
          software only supports vpci 0, although any  value  may
          be  placed  in  the vpci field and will be encoded into
          the message.

     qcc_qos_t
          The Quality of Service IE has 3  informational  fields:
          codingstd,  specifying  the  Coding Standard value; and
          forward_class and backward_class, specifying  the  For-
          ward and Backward QoS Class.

     qcc_restart_ind_t
          There is only one informational field  in  the  Restart
          Indicator  IE  structure:  class,  whcih  specifies the
          class of the facility to be restarted.

     qcc_transit_t
          The Transit Network Selection IE structure contains  an
          array of up to four octets to specify the Carrier Iden-
          tification Code value.

     qcc_endpt_ref_t
          The  Endpoint  Reference  IE  structure   contains   an
          endptref  field, which specifies the endpoint reference
          value.

     qcc_endpt_state_t
          The Endpoint State IE structure contains a state field,
          which identifies the endpoint state value.

RETURN VALUES
     The function returns 0 on success and -1 on error.

EXAMPLES
     See the Example section of the qcc_create(3) man page for an
```

CODE EXAMPLE 2-7 `qcc_set_ie(3)` Man Page *(Continued)*

```
     example using qcc_set_ie.

SEE ALSO
     qcc_create(3),  qcc_pack(3),  qcc_unpack(3),   qcc_parse(3),
     qcc_util(3), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.
```

qcc_unpack(3)

CODE EXAMPLE 2-8 qcc_unpack(3) Man Page

```
qcc_unpack(3)          C Library Functions          qcc_unpack(3)



NAME
     qcc_unpack,      qcc_unpack_setup,      qcc_unpack_alerting,
     qcc_unpack_call_proceeding,              qcc_unpack_connect,
     qcc_unpack_connect_ack,                  qcc_unpack_release,
     qcc_unpack_release_complete,              qcc_unpack_status,
     qcc_unpack_status_enq,                    qcc_unpack_notify,
     qcc_unpack_restart,                  qcc_unpack_restart_ack,
     qcc_unpack_add_party,              qcc_unpack_add_party_ack,
     qcc_unpack_party_alerting,      qcc_unpack_add_party_reject,
     qcc_unpack_drop_party,            qcc_unpack_drop_party_ack,
     qcc_unpack_leaf_setup_fail,    qcc_unpack_leaf_setup_req   -
     decode Q.2931 messages and unpack into message structures

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/types.h>
     #include <atm/qcc.h>

     int qcc_unpack_setup(qcc_setup_t *msgp, strbuf_t *ctlp,
           strbuf_t *datap);

     int qcc_unpack_alerting(qcc_alerting *msgp, strbuf_t *ctlp,
           strbuf_t *datap);

     int qcc_unpack_call_proceeding(qcc_call_proc_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_connect(qcc_connect_t *msgp, strbuf_t *ctlp,
           strbuf_t *datap);

     int qcc_unpack_connect_ack(qcc_connect_ack_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_release(qcc_release_t *msgp, strbuf_t *ctlp,
           strbuf_t *datap);
```

CODE EXAMPLE 2-8 `qcc_unpack(3)` Man Page *(Continued)*

```
     int qcc_unpack_release_complete(qcc_release_complete_t *
           msgp, strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_status_enq(qcc_status_enq_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_status(qcc_status_t *msgp, strbuf_t *ctlp,
           strbuf_t *datap);

     int qcc_unpack_notify(qcc_notify_t *msgp, strbuf_t *ctlp,
           strbuf_t *datap);

     int qcc_unpack_restart(qcc_restart_t *msgp, strbuf_t *ctlp,
           strbuf_t *datap);

     int qcc_unpack_restart_ack(qcc_restart_ack_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_add_party(qcc_add_party_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_add_party_ack(qcc_add_party_ack_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_party_alerting(qcc_party_alerting_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_add_party_reject(qcc_add_party_reject_t *
           msgp, strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_drop_party(qcc_drop_party_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_drop_party_ack(qcc_drop_party_ack_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_leaf_setup_fail(qcc_leaf_setup_fail_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

     int qcc_unpack_leaf_setup_req(qcc_leaf_setup_req_t *msgp,
           strbuf_t *ctlp, strbuf_t *datap);

MT-LEVEL
     Safe.
```

CODE EXAMPLE 2-8 qcc_unpack(3) Man Page *(Continued)*

```
AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions  take  streams  buffers  containing  encoded
     Q.2931 messages as input and decode the information, placing
     the extracted values into the appropriate message structure.
     The  Q.2931  protocol  is  used  for  ATM signalling; a full
     description of the message format and use can  be  found  in
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0. Messages conforming to both versions  of  the
     UNI  standard will be decoded.  The functions may be used by
     processes which are running in user space.

     In general, no error checking is performed on the data  that
     is  extracted from the message.  Whatever data is found will
     be placed in the message structure without examination.

     Each function  requires  3  parameters:  msgp,  which  is  a
     pointer  to  the appropriate message structure; and ctlp and
     datap, which are pointers to strbuf_t buffers.

     ctlp  is  the  control  portion  of  a   received   message,
     corresponding  to  the  M_CTL  block of the message that was
     received from downstream.  datap is the data portion of  the
     message, corresponding to the M_DATA block.

     The message structure pointed to by msgp should be allocated
     by the user program which calls a qcc_unpack function.

RETURN VALUES
     All functions return 0 on  success  and  -1  on  error.  The
     returned message structure contains an entry for each possi-
     ble Information Element for that message type; if an  Infor-
     mation Element is found in the received message, the "valid"
     field for that IE will be set to  1.   If  the  IE  was  not
     found, the "valid" field will be 0.

EXAMPLES
     The following code fragment receives  a  setup  message  and
     prints elements in the message structure.

          #include <atm/types.h>
```

CODE EXAMPLE 2-8 qcc_unpack(3) Man Page *(Continued)*

```
          #include <atm/qcc.h>
          #include <atm/limits.h>

          void
          wait_for_setup(int fd);
          {
               int            flags = 0;
               int            vci = -1;
               int            sap = -1;
               qcc_hdr_t      *hdrp;
               qcc_setup_t    setup;
               struct strbuf  ctl, data;
               char           ctlbuf[QCC_MAX_CTL_LEN];
               char           databuf[QCC_MAX_DATA_LEN];

               ctl.buf = ctlbuf;
               data.buf = databuf;
               ctl.len = data.len = 0;
               ctl.maxlen = QCC_MAX_CTL_LEN;
               data.maxlen = QCC_MAX_DATA_LEN;

               if (getmsg(fd, &ctl, &data, &flags) < 0) {
                    perror("getmsg");
                    exit (-1);
               }

               hdrp = qcc_get_hdr(&ctl);
               if ((hdrp) && (hdrp->type == QCC_SETUP)) {
                    if ((qcc_unpack_setup(&setup, &ctl, &data)) < 0) {
                         printf("parse_setup failed\n");
                         exit (-1);
                    }
                    if (setup.conn_id.valid)
                         vci = setup.conn_id.vci;
                    if (setup.bhli.valid)
                         memcpy((caddr_t) &sap,
                                (caddr_t) setup.bhli.info, 4);

                    printf("parse_setup: vci=0x%x, sap=0x%x\n",
                            vci, sap);
               }
          }

SEE ALSO
     qcc_len(3),   qcc_create(3),   qcc_set_ie(3),   qcc_pack(3),
     qcc_util(3), q93b(7)
```

CODE EXAMPLE 2-8 `qcc_unpack(3)` Man Page *(Continued)*

```
     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message types will be ignored by the
     q93b driver if used on an interface which is configured  for
     UNI 3.0 or 3.1.  The UNI 4.0-specific messages are Alerting,
     Notify, Party Alerting, Leaf  Setup  Fail,  and  Leaf  Setup
     Request.
```

qcc_util(3)

CODE EXAMPLE 2-9 qcc_util(3) Man Page

```
qcc_util(3)            C Library Functions           qcc_util(3)



NAME
     qcc_util, q_ioc_bind, q_ioc_bind_lijid, q_ioc_unbind_lijid -
     functional interfaces to q93b driver ioctls

SYNOPSIS
     cc [ flag ... ] file ... -latm [ library ... ]

     #include <atm/qcc.h>

     int q_ioc_bind(int fd, int sap);

     int q_ioc_bind_lijid(int fd, int lijid);

     int q_ioc_unbind_lijid(int fd, int lijid);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The libatm.a library, which is located in /usr/lib, must  be
     included at compile time as indicated in the synopsis.

DESCRIPTION
     These functions may be used to provide information about the
     user application to the q93b driver.

     Before using these functions, a stream must be opened to the
     q93b driver, using the open(2) system call.

     q_ioc_bind() binds a service access point, sap, to an opened
     stream,  specified  by its file descriptor, fd. This step is
     required so that incoming SETUP messages are directed to the
     correct  application  by  the q93b driver. Q.2931 SETUP mes-
     sages which are to be received by  the  application  program
```

CODE EXAMPLE 2-9 qcc_util(3) Man Page *(Continued)*

```
     must contain a Broadband Higher Layer Information IE identi-
     fying the sap to which the message should be directed.

     q_ioc_bind_lijid() binds a leaf-initiated join id, lijid, to
     an  opened  stream,  specified  by  its file descriptor, fd.
     This functionality is in support of a  new  feature  in  UNI
     4.0,  which  allows  endpoints  to  request  to  be added to
     specific point-to-multipoint calls, identified by the  leaf-
     initiated  join  id.   An  application that wishes to be the
     root of a  point-to-multipoint  call  which  supports  leaf-
     initiated  join  must  associate  its  q93b  stream with the
     call's leaf-initiated join id in one of two ways: by setting
     up  a call in which the leaf-initiated join id is specified,
     or by calling this function.

     q_ioc_unbind_lijid() breaks the association between a  leaf-
     initiated  join  id,  lijid,  and a stream, specified by its
     file descriptor, fd.

RETURN VALUES
     The functions return 0 on success and -1 on error.

EXAMPLES
     The following example opens a stream to q93b and binds it to
     sap 0x100.

          #include <atm/qcc.h>

          setup_q93b();
          {
               char      qdriver[] = "/dev/q93b";
               int       qfd;
               int       sap = 0x100;

               if ((qfd = open(qdriver, O_RDWR, 0)) < 0) {
                    perror("open");
                    exit(-1);
               }

               if (q_ioc_bind(qfd, sap) < 0) {
                    perror("q_ioc_bind");
                    exit(-1);
               }
          }

SEE ALSO
```

CODE EXAMPLE 2-9 `qcc_util(3)` Man Page *(Continued)*

```
atm_util(3),    qcc_bld(3),    qcc_create(3),    qcc_len(3),
qcc_pack(3),   qcc_parse(3),   qcc_unpack(3),   qcc_bld(9F),
qcc_create(9F),  qcc_len(9F),  qcc_pack(9F),  qcc_parse(9F),
qcc_unpack(9F), q93b(7), ba(7)
```

CHAPTER 3

File Formats

The man pages in this chapter describe the configuration files in the SunATM software.

TABLE 3-1 File Format Man Pages

Man Page	Description	Page Number
`aarconfig(4)`	ATM Address Resolver configuration file	page 76
`acl.cfg(4)`	SunATM SNMP access-privileges database group configuration file	page 85
`agent.cnf(4)`	SunATM SNMP agent configuration file	page 87
`atmconfig(4)`	SunATM interface configuration file	page 89
`context.cfg(4)`	SunATM SNMP contexts database group configuration file	page 91
`ilmi.cnf(4)`	SunATM SNMP agent configuration file for `ilmid(1M)`	page 94
`laneconfig(4)`	LAN Emulation configuration file	page 95
`mib.rt(4)`	SunATM SNMP agent utility file	page 103
`party.cfg(4)`	SunATM SNMP party database group configuration file	page 105
`view.cfg(4)`	SunATM SNMP MIB-view database group configuration file	page 108

aarconfig(4)

CODE EXAMPLE 3-1 aarconfig(4) Man Page

```
aarconfig(4)              File Formats              aarconfig(4)



NAME
     aarconfig - ATM Address Resolver configuration file

SYNOPSIS
     /etc/aarconfig

DESCRIPTION
     The aarconfig file is a local database that  associates  ATM
     addresses  with  IP  addresses.  The file is used by the ATM
     Address  Resolution  setup  program,   aarsetup(1M),   which
     manages  the  downloading of local information into the ker-
     nel. If changes are made to the aarconfig file, aarsetup(1M)
     must be rerun for the changes to take effect.

     If an ATM ARP server does not exist on a subnet,  an  ATM/IP
     address  pair  must  appear in each system's local aarconfig
     file in order for the system to communicate with that node.

     An ATM ARP server solves the problem of having to explicitly
     enter  ATM/IP  address pairs into a table at each node. When
     client interfaces  come  up,  they  register  with  the  ARP
     server,  which  then  sends  an  inverse  ARP request to the
     client. The client responds with its IP address; the  server
     then  enters the information into its kernel-resident table.
     Clients may then resolve addresses with  the  server,  using
     ARP  requests.  If an ATM ARP server is being used in a sub-
     net, clients only need local information and server informa-
     tion in their own configuration files.

     The format of an entry in aarconfig is:

          Interface    Hostname    ATM-Address   VC    Flags

     Items are separated by any number of SPACE and/or TAB  char-
     acters.   The  first  item  is the physical interface on the
     local system which is attached to the subnet for this entry.
```

CODE EXAMPLE 3-1 `aarconfig(4)` Man Page *(Continued)*

```
     It  should  be of the form "device unit;" an example is ba0.
     Hostname can be an IP hostname or address  in  the  standard
     dot  notation.   The  ATM  address is a 20 byte address; its
     format is hexadecimal bytes (2 characters) separated by  one
     or  more colons (additional colons may be used for readabil-
     ity, if desired).  The VC field specifies the  virtual  con-
     nection  identifier  (VCI)  for  the  connection to the host
     identified by this entry.  The flag field gives  information
     regarding  the  type  of  entry.  Comment lines are allowed;
     they are indicated by a `#' at the beginning of the line.

     ATM addresses are 20 bytes.  The first 13 bytes (called  the
     prefix) are used by the switch for routing purposes; in gen-
     eral, they will be the same for addresses connected  to  the
     same  switch.  The prefix is assigned by the switch and will
     be sent to the host during address  registration  (performed
     by  ILMI)  when the ATM interface on the host system is con-
     figured.  The predefined variable  `prefix'  (see  Variables
     section  below)  will  be assigned the value received by the
     host from the switch at configuration time; this  value  may
     be referenced in the aarconfig file as `$prefix'.

     The next 6 bytes (called the ESI) are used to uniquely iden-
     tify  a  host  system;  in  most  of the examples given, the
     system's hardware MAC address is used. The MAC  address  may
     be  referenced  in  the  aarconfig file as `$mac'. The final
     byte is a selector byte that may be used  by  the  host  for
     internal  routing  of  data.  Use of the predefined variable
     `sel' will guarantee that an appropriate value for the given
     interface will be used.

     Depending on the entry type,  as  determined  by  the  flags
     field,  some or all of the fields are required.  All entries
     must have an  interface  and  flags  field;  the  host,  atm
     address,  and VC field vary depending on the entry type.  An
     entry should never have both an ATM address field and  a  VC
     field;  an  ATM address indicates that Switched Virtual Cir-
     cuits (SVCs) should be used for connections, and a VC  indi-
     cates that Permanent Virtual Circuits (PVCs) should be used.
     The following section defines  each  flag  type,  and  lists
     which  of  the host, atm address, and VC fields are required
     for that type. An empty  field  should  be  indicated  by  a
     hyphen `-'.

OPTIONS
  Variables
```

CODE EXAMPLE 3-1 aarconfig(4) Man Page *(Continued)*

```
     Because the prefix portion of an ATM address  specifies  the
     ATM switch, a number of hosts specified in an aarconfig file
     may have ATM addresses who share the same  prefix.  To  sim-
     plify  setting  up  the aarconfig file, one can define vari-
     ables that contain part of an ATM address. A variable's name
     is  an identifier consisting of a collection of no more than
     32 letters, digits, and underscores (`_'). The value associ-
     ated  with  the  variable  is denoted by a dollar sign (`$')
     followed immediately by the variable name.

     Variables may only be used in the ATM address  field.   They
     may not be used in any of the other fields in an entry.

     Multiple variables may be concatenated to represent a single
     ATM  address expression. A colon must be used to concatenate
     the variables.  Thus, if one variable, v1, is set to `11:22'
     and  another,  v2,  is  set to `33:44', the sequence $v1:$v2
     represents `11:22:33:44'. Hexadecimal numbers  may  also  be
     included  with  variables  in the expression. The expression
     `45:$v1:$v2' would have the value `45:11:22:33:44'.

     Variables are defined in the aarconfig file according to the
     following format:

          set VARIABLE = EXPRESSION

     where VARIABLE is the name of a variable and  EXPRESSION  is
     an  expression  concatenating  one- or two-digit hexadecimal
     numbers and/or the values of variables that have been previ-
     ously defined.  The equal sign is optional, but the variable
     and  expression  must  be  separated  by  either  whitespace
     (spaces or tabs), an equal sign, or both.

     Several predefined variables are  built  in  to  the  SunATM
     software.  They include:

     prefix       the 13-byte prefix associated  with  the  local
                  switch.

     mac          the 6-byte  MAC  address  associated  with  the
                  local host or interface.

     sel          the  default  1-byte  Selector  for  the  local
                  interface.

     macsel       the concatenation of $mac:$sel.
```

CODE EXAMPLE 3-1 aarconfig(4) Man Page *(Continued)*

```
myaddress    the concatenation of $prefix:$mac:$sel, result-
             ing in the default address for the local inter-
             face.

anymac       a wild card representing any 6-byte ESI. Should
             only be used in `a' entries.

anymacsel    a wild card representing  any  7-byte  ESI  and
             Selector  combination.   Should only be used in
             `a' entries.

sunmacselN   the  concatenation  of  one  of  a  series   of
             reserved  MAC  addresses  and  $sel to create a
             block of reserved 7-byte ESI and Selector  com-
             binations  which  may be used in ATM ARP server
             addresses. N should be a decimal number in  the
             range 0 - 199.

localswitch_server
             the concatenation of $prefix, a unique reserved
             MAC  address,  and  $sel. When used as a server
             address, restricts  server  access  to  clients
             connected to the local switch only.

In most network configurations, the ATM address assigned  to
the  local  interface will be myaddress; using this variable
in the `l' entry makes it possible to use identical  aarcon-
fig files on all clients using a given server.

The sunmacselN variables may be used  to  create  well-known
server addresses which are not bound to a particular system.
The prefix portion is not included so the addresses  may  be
used  on  systems  connected  to different switches. The ESI
portion of a sunmacselN  variable  is  one  of  a  range  of
reserved    MAC    addresses.    The    base    address    is
08:00:20:75:48:10; to calculate the MAC address for any sun-
macselN  variable, simply add the value of N (converted to a
hexadecimal number) to the base address.  For  example,  the
ESI portion of sunmacsel20 would be 08:00:20:75:48:10 + 0x14
= 08:00:20:75:48:24.

Finally, localswitch_server may  be  used  as  a  well-known
server  address  in  an  isolated net, that is, one in which
server access is restricted to clients on the local  switch.
Thus  any  host with a network prefix other than that of the
```

CODE EXAMPLE 3-1 aarconfig(4) Man Page *(Continued)*

```
       local switch will be refused a connection to the ARP  server
       if  the  ARP server's address is localswitch_server. The ESI
       portion of localswitch_server is the  reserved  MAC  address
       08:00:20:75:48:08.

       Several rules apply to the use of variables in the aarconfig
       file:

            Two variables cannot follow each other in an expression
            without  an  intervening  colon. Thus, $v1:$v2 is legal
            whereas $v1$v2 is not.

            Fields in each line in the aarconfig file are separated
            by  whitespace.   Therefore  variables  should  not  be
            separated from the rest of an  ATM  address  with  whi-
            tespace. For example, $v1: $v2 is illegal.

            Once a variable is defined by a set command, it may not
            be redefined later in the aarconfig file.

            The reserved  variable  names  may  not  be  set.  They
            include  `prefix', `mac', `sel', `macsel', `myaddress',
            `anymac',  `anymacsel',  `sunmacselN'  (where  N  is  a
            number between 0 and 199), and `localswitch_server'.

    Basic Configuration Flags
       l    This flag identifies an entry for a local interface  on
            an  ARP  client  or  system  that  does  not use an ARP
            server.

            If SVCs are to be used at all on  this  interface,  the
            ATM  address  is  required;  an empty ATM address field
            indicates PVCs only on this interface.  The host should
            not  be  entered;  the  system will locate the hostname
            assigned to this physical interface.  No VC  should  be
            entered  either, since there will typically be multiple
            VCs over the local interface.

       L    This flag identifies an entry for a local interface  on
            an ARP server.

            The ATM address is required. No host or  VC  should  be
            entered.

       t    Adds this host to the local table.
```

CODE EXAMPLE 3-1 aarconfig(4) Man Page *(Continued)*

```
             The host is required; either an ATM  address  or  a  VC
             field  is required, depending on whether a SVC or a PVC
             connection is desired.  If a mixture  of  SVC  and  PVC
             connections  is  desired,  both an ATM address and a VC
             are allowed.

        s    Specifies a connection to the  ATM  ARP  Server.   This
             identifies  to  the ARP client where it should make ARP
             (address resolution) requests for  addresses  that  are
             not in its local table.

             Either the atm address in the case of a SVC connection,
             or  the  VC  in  the  case  of a PVC connection, should
             appear (but not both); the host should not appear.

        The required, optional, and illegal  fields  for  the  basic
        flag types are summarized in the following table:

        -----------------------------------------------
        Interface   Host      ATM-Addr   VCI     FLAGS
        -----------------------------------------------
        required    illegal   optional   illegal   l
        required    illegal   required   illegal   L
        required    required  or         or*       t
        required    illegal   xor        xor**     s
        -----------------------------------------------
        * one or the other is required, but both are also legal.
        ** one or the other is required; both are illegal.

     Advanced Configuration Flags
        The basic configuration flags are sufficient for most  stan-
        dard  network  configurations.   However, since networks are
        rarely homogeneous, there may be cases in which, for intero-
        perability  purposes, a network must be configured with dif-
        ferent characteristics than the defaults that are built into
        the  SunATM adapter, or with unusual addressing schemes that
        require more than the basic  configuration  flags  described
        above.   The following flags may also be used in the aarcon-
        fig file to alter the default behavior when necessary.

        a    On an ARP server, represents an ATM  address  that  may
             have  access  to  this  ARP  server.  If no `a' entries
             appear in the server's aarconfig file, any ATM host may
             register  with  the  ARP  server. Including `a' entries
             restricts access to known hosts. The wildcard variables
             described   in   the   variable  section  (`anymac'  and
```

CODE EXAMPLE 3-1 aarconfig(4) Man Page *(Continued)*

```
          `anymacsel') may be used to  specify  groups  of  hosts
          connected  to a common switch to be allowed access in a
          single entry, or specific addresses may  listed.  NOTE:
          If  this  value  is  changed, only a reboot will ensure
          that old addresses are not being cached.

          The host and VC should not appear; an  ATM  address  is
          required.

     m    Specifies manual address configuration mode. This indi-
          cates  to the system that ILMI is not being used on the
          specified interface.  Entries for  non-ILMI  interfaces
          may  not  use  the $prefix variable, or variables which
          make  use  of   $prefix   (such   as   $myaddress   and
          $localswitch_server),  since  ilmid will not be able to
          provide this information.

          Only the interface is required. The  MAC  address,  ATM
          address, and VCI should not appear.

     The required, optional, and illegal fields for the  advanced
     flag types are summarized in the following table:

     -----------------------------------------------
     Interface   Host      ATM-Addr   VCI     FLAGS
     -----------------------------------------------
     required    illegal   required   illegal   a
     required    illegal   illegal    illegal   m
     -----------------------------------------------

EXAMPLES
     The following lines show the simplest case  aarconfig  files
     for  a  single-switch  network  in which ARP clients use the
     default address  for  their  interface  and  all  hosts  are
     allowed access to the server:


          in the client's aarconfig:

               ba0 - $myaddress - l
               ba0 - $localswitch_server - s

          in the server's aarconfig:

               ba0 - $localswitch_server - L
```

CODE EXAMPLE 3-1 `aarconfig(4)` Man Page *(Continued)*

```
     The following line defines the local interface  for  an  ARP
     client  which does not use the local MAC address for its ESI
     on its ba1 port:

          ba1 - $prefix:08:00:20:1a:e1:53:$sel - l

     The following lines would be placed in the  aarconfig  files
     on two machines connected back-to-back over PVC.


          in the aarconfig of host1:

               ba0 - - - l
               ba0 host2 - 100 t

          in the aarconfig of host2:

               ba0 - - - l
               ba0 host1 - 100 t

     The following lines would be placed in the aarconfig file on
     a  server to restrict access to those hosts connected to the
     local switch or an explicitly identified remote switch.  The
     server is using a predefined server address.


          set remote = 45:00:00:00:00:00:00:00:0f:01:02:03:04

          ba0 - $prefix:$sunmacsel0 - L

          ba0 - $prefix:$anymacsel - a

          ba0 - $remote:$anymacsel - a

SEE ALSO
     aarsetup(1M)

     M. Laubach, RFC 1577: Classical IP and ARP over ATM, Network
     Working Group.

NOTES
     In the current implementation, the entries must  be  grouped
     by  type and in a particular order: the local (l or L) entry
     should be first, then the table (t) entries (if  used),  and
     finally  server (s) entries.  Other flag types may appear in
     any order.  Also, the ordering need only be maintained among
```

CODE EXAMPLE 3-1 `aarconfig(4)` Man Page *(Continued)*

```
     entries for each physical interface; for example, all of the
     ba0 entries may appear  first,  and  then  all  of  the  ba0
     entries.   This requirement will likely be relaxed in future
     releases.

     Each entry should be entered on one line with no  breaks  or
     carriage returns.
```

acl.cfg(4)

CODE EXAMPLE 3-2 acl.cfg(4) Man Page

```
acl.cfg(4)                File Formats                acl.cfg(4)



NAME
     acl.cfg - SunATM SNMP access-privileges database group  con-
     figuration file

SYNOPSIS
     /etc/opt/SUNWatm/snmp/acl.cfg

DESCRIPTION
     The acl.cfg file contains the access-privileges database for
     the  SunATM  SNMP agent, amtsnmpd(1M). The entries contained
     in this file are the conceptual rows of  the  aclTable  (RFC
     1447).

     Each conceptual row contains the following entries:

     aclTarget      The SNMPv2 party which is the  target  of  an
                    access control policy.

     aclSubject     The SNMPv2 party which is the subject  of  an
                    access control policy.

     aclResources   The SNMPv2 context in an access control  pol-
                    icy.

     aclPrivileges  An  integer  in  the  range  of  0-255  which
                    specify what management operations a particu-
                    lar target party may perform with respect  to
                    a particular context when requested by a par-
                    ticular subject party. These  privileges  are
                    specified  as  a  sum  of  values, where each
                    value specifies a SNMPv2 PDU  type  by  which
                    the  subject  party  may  request a permitted
                    operation.

     aclStorageType The storage type for this conceptual  row  in
                    the aclTable. Takes on the values 1-4.
```

CODE EXAMPLE 3-2 acl.cfg(4) Man Page *(Continued)*

```
     aclStatus      The status of  this  conceptual  row  in  the
                    aclTable.  Takes  on the values valid (1) and
                    invalid (2).

     Each entry in the file is represented by 5 lines.

           aclTarget
           aclSubject
           aclResources
           aclStatus
           aclStorageType (decimal) aclPrivileges (hex)

     Symbolic names may be used as long as  they  appear  in  the
     mib.rt(4)  file.  Otherwise  the  dotted  object ids must be
     used.  ';' is the comment character.  Comments may not be in
     between sections of an acl.

EXAMPLES
     The following is an example of a typical acl  entry  in  the
     acl.cfg file.

           initialPartyId.127.0.0.1.1
           initialPartyId.127.0.0.1.2
           initialContextId.127.0.0.1.1
           001
           003 002b

     This entry defines the aclTarget,  the  aclSubject  and  the
     aclContext  for  this  aclEntry,  as well as an aclStatus of
     active (1), aclStorageType nonVolatile (3) and aclPrivileges
     Get, GetNext, GetBulk and Set (2b).

SEE ALSO
     atmsnmpd(1M),  view.cfg(4),  party.cfg(4),   context.cfg(4),
     mib.rt(4)
```

agent.cnf(4)

CODE EXAMPLE 3-3 agent.cnf(4) Man Page

```
agent.cnf(4)              File Formats              agent.cnf(4)



NAME
     agent.cnf - SunATM SNMP agent configuration file

SYNOPSIS
     /etc/opt/SUNWatm/snmp/agent.cnf

DESCRIPTION
     The agent.cnf file defines basic  configuration  information
     for the SunATM SNMP agent, amtsnmpd(1M).

     Each entry contains  a  keyword,  followed  by  a  parameter
     string.   The keyword should be in the first position in the
     line, and an entry must be contained in a single  line.  The
     keyword  may  be  separated  from  parameters  by whitespace
     (spaces or tabs), and comments are denoted by a '#'  charac-
     ter.

OPTIONS
     The following list contains  the  currently  supported  key-
     words.

     syscontact        The value to be used to answer queries for
                       sysContact.

     syslocation       The value to be used to answer queries for
                       sysLocation.

     trap              A list of hosts which should receive traps
                       (one or more hosts may be included).

     read-community    The community name which should have  read
                       access.

     write-community   The community name which should have write
                       access.  Write access implies read access.
```

CODE EXAMPLE 3-3 `agent.cnf(4)` Man Page *(Continued)*

```
     trap-community    The community name to be used in traps.

SEE ALSO
     atmsnmpd(1M)
```

atmconfig(4)

CODE EXAMPLE 3-4 atmconfig(4) Man Page

```
atmconfig(4)             File Formats             atmconfig(4)



NAME
     atmconfig - SunATM interface configuration file

SYNOPSIS
     /etc/atmconfig

DESCRIPTION
     The atmconfig file is a  local  database  that  defines  the
     feature  set required for each SunATM interface in a system.
     The file is used by the /etc/rc2.d/S00sunatm  script,  which
     runs  at  boot  time  to  configure  SunATM  interfaces.  If
     changes are made to the atmconfig file, the system  must  be
     rebooted for the changes to take effect.

     The format of an entry in atmconfig is:

            Physical     UNI Ver/    C-IP    LANE    LANE
            Interface    Framing     Host    Inst    Host

     Items are separated by any number of SPACE and/or TAB  char-
     acters.   The  first  item  is the physical interface on the
     local system.  It should be of the form  "device  unit;"  an
     example  is ba0.  UNI Version is the UNI version number that
     should be used on this interface; SunATM  2.1  supports  3.0
     and 3.1.  This field can also be used to specify the framing
     interface to be used on a particular SunATM physical  inter-
     face;  both  the SONET and SDH protocols are supported.  The
     default framing is sonet unless /etc/system indicates other-
     wise.  The third field is the Classical IP hostname for this
     interface, if Classical IP is to be run on  this  interface.
     The  fourth and fifth fields are used if LAN Emulation is to
     be run on this interface; these fields are the LAN Emulation
     instance  number  (each  LAN Emulation interface must have a
     unique number, and interfaces will  appear  in  ifconfig  as
     laneN,  where N is the instance number), and the IP hostname
     for the LAN Emulation interface.
```

CODE EXAMPLE 3-4 `atmconfig(4)` Man Page *(Continued)*

```
     Depending on the IP protocols to be supported, some  or  all
     of  the  fields are required.  Every interface that is to be
     configured must have at least one  entry  in  /etc/atmconfig
     which  contains  a minimum of the interface name and the UNI
     version.  In addition, the Classical IP Hostname is required
     if  Classical IP (RFC 1577) is to be supported; and the LANE
     Instance is required if LAN Emulation is  to  be  supported.
     Further entries for the same interface may be included after
     the entry containing the UNI version to specify multiple LAN
     Emulation  instances,  multiple  logical  interfaces  or the
     framing.  Refer to Chapter 5 in the SunATM  2.1  Manual  for
     further information on multiple entries.  In all entries, an
     empty field should be indicated by a hyphen `-'.

EXAMPLES
     The following example shows the atmconfig file for a  system
     with  three SunATM interfaces.  The first, ba0, supports UNI
     3.1 and LAN Emulation.  The second, ba1,  supports  UNI  3.1
     and  both  Classical IP and LAN Emulation.  The third inter-
     face, ba2, supports UNI 3.0 with Classical IP and  uses  SDH
     framing.


          #
          ba0       3.1       -         0         atm0
          #
          ba1       3.1       atm1      1         atm2
          #
          ba2       3.0       atm3      -         -
          ba2       SDH       -         -         -

SEE ALSO
     aarconfig(4), laneconfig(4)

NOTES
     Each entry should be entered on one line with no  breaks  or
     carriage returns.
```

context.cfg(4)

CODE EXAMPLE 3-5 context.cfg(4) Man Page

```
context.cfg(4)            File Formats             context.cfg(4)



NAME
     context.cfg - SunATM SNMP contexts database group configura-
     tion file

SYNOPSIS
     /etc/opt/SUNWatm/snmp/context.cfg

DESCRIPTION
     The context.cfg file contains the contexts database for  the
     SunATM  SNMP  agent,  amtsnmpd(1M). The entries contained in
     this file are the conceptual rows of the  contextTable  (RFC
     1447).

     Each conceptual row contains the following entries:

     contextIdentity      A context identifier uniquely identify-
                          ing a particular SNMPv2 context.

     contextLocal         An indication of whether  this  context
                          is  realized  by  this  SNMPv2  entity.
                          Takes on the values true (1)  or  false
                          (2).

     contextStorageType   The storage type of this conceptual row
                          in   the  contextTable.  Takes  on  the
                          values 1-4.

     contextStatus        The status of this  conceptual  row  in
                          the  contextTable.  Takes on the values
                          valid (1) and invalid (2).

     contextViewIndex     If zero, this row refers to  a  context
                          which  identifies a proxy relationship;
                          otherwise, this row refers to a context
                          that identifies a MIB view of a locally
                          accessible entity.
```

CODE EXAMPLE 3-5 context.cfg(4) Man Page *(Continued)*

```
     contextLocalEntity   If  contextViewIndex  is  greater  than
                          zero,  this  value identifies the local
                          entity whose management information  is
                          in  this  context's MIB view. The empty
                          string indicates that the MIB view con-
                          tains the entity's own local management
                          information.

     contextLocalTime     If  contextViewIndex  is  greater  than
                          zero,  this  value  identifies the tem-
                          poral context of the management  infor-
                          mation in the MIB view.

     contextProxyDstParty If contextViewIndex is equal  to  zero,
                          this  value  identifies a party that is
                          the proxy destination of a proxy  rela-
                          tionship.

     contextProxySrcParty If contextViewIndex is equal  to  zero,
                          this  value  identifies a party that is
                          the proxy source of a  proxy  relation-
                          ship.

     contextProxyContext  If contextViewIndex is equal  to  zero,
                          this  value identifies the context of a
                          proxy relationship.

     Each entry in the file is represented by 8 lines.

           contextIdentity
           contextStatus
           contextLocal contextStorageType contextViewIndex
           contextLocalEntity
           contextLocalTime
           contextProxyDstParty
           contextProxySrcParty
           contextProxyContext


     Symbolic names may be used as long as  they  appear  in  the
     mib.rt(4)  file.  Otherwise  the  dotted  object ids must be
     used.  ';' is the comment character.  Comments may not be in
     between sections of a context.

EXAMPLES
```

CODE EXAMPLE 3-5 `context.cfg(4)` Man Page *(Continued)*

```
     The following is an example of a typical  context  entry  in
     the context.cfg file.

           initialContextId.127.0.0.1.1
           001
           001 003 00001
           <empty line>
           currentTime
           <empty line>
           <empty line>
           <empty line>

     This entry defines the contextIdentity object identifier for
     the  specific  contextEntry,  contextStatus active (1), con-
     textLocal true (1), contextStorageType nonVolatile (3), con-
     textViewIndex  the  viewEntry  with  viewIndex 1, contextLo-
     calEntity with value the empty string, contextLocalTime with
     obcect  identifier currentTime refering to management infor-
     mation at the current time, and no entries for  contextProx-
     ySrcParty,   contextProxySrcParty   and  contextProxyContext
     (empty lines).

SEE ALSO
     atmsnmpd(1M),   view.cfg(4),    party.cfg(4),    acl.cfg(4),
     mib.rt(4)
```

ilmi.cnf(4)

CODE EXAMPLE 3-6 ilmi.cnf(4) Man Page

```
ilmi.cnf(4)              File Formats               ilmi.cnf(4)



NAME
     ilmi.cnf  -  SunATM  SNMP  agent  configuration   file   for
     ilmid(1M).

SYNOPSIS
     /etc/opt/SUNWatm/snmp/ilmi.cnf

DESCRIPTION
     The  ilmi.cnf  file  defines  the  community  name  used  by
     ilmid(1M)  to  send  requests  to  the  SunATM  SNMP  agent,
     atmsnmpd(1M).

     Each entry consists of a keyword  followed  by  a  parameter
     string.   The keyword should be in the first position in the
     line, and an entry must be contained in a single  line.  The
     keyword  may  be  separated  from  parameters  by whitespace
     (spaces or tabs), and comments are denoted by a '#'  charac-
     ter.

OPTIONS
     The following list contains  the  currently  supported  key-
     words.

     ilmi-community    The  community  name   to   be   used   by
                       ilmid(1M).

SEE ALSO
     atmsnmpd(1M)
```

laneconfig(4)

CODE EXAMPLE 3-7 `laneconfig(4)` Man Page

```
laneconfig(4)            File Formats             laneconfig(4)



NAME
     laneconfig - LAN Emulation configuration file

SYNOPSIS
     /etc/laneconfig

DESCRIPTION
     The laneconfig file is a local database that associates  MAC
     addresses  with  ATM addresses.  The file is used by the LAN
     Emulation setup program, lanesetup(1M),  which  manages  the
     downloading  of the information found in laneconfig into the
     kernel.  If  changes  are  made  to  the  laneconfig   file,
     lanesetup(1M) must be rerun for the changes to take effect.

     The format of an entry in laneconfig is:

          Interface    MAC-Address/    ATM-Address   VC   Flags
                        ELAN Name

     Items are separated by any number of SPACE and/or TAB  char-
     acters.   The  first  item is the LAN Emulation interface on
     the local system which is attached to the  subnet  for  this
     entry.   It should be of the form "lane unit;" an example is
     lane0.  The MAC address is the 6 byte physical MAC  address;
     it should be specified as 6 hexadecimal bytes (2 characters)
     separated by one or more colons (additional  colons  may  be
     used  for  readability,  if  desired).  In some entries, the
     second field will be an Emulated LAN name, which is a  char-
     acter string. The ATM address is a 20 byte address; its for-
     mat is the same colon-separated hexadecimal format used  for
     the MAC address.  The VC field specifies the virtual connec-
     tion identifier (VCI) for the connection to the host identi-
     fied  by  this  entry.   The  flag  field  gives information
     regarding the type of entry.   Comment  lines  are  allowed;
     they are indicated by a `#' at the beginning of the line.
```

CODE EXAMPLE 3-7 laneconfig(4) Man Page *(Continued)*

```
     ATM addresses are 20 bytes.  The first 13 bytes (called  the
     prefix)  are  used  by the switch for routing purposes.  The
     prefix is assigned by the switch and will  be  sent  to  the
     host  when  the  ATM interface on the host system is config-
     ured.  The predefined variable `prefix' (see Variables  sec-
     tion  below) will be assigned the value received by the host
     from the switch at configuration time;  this  value  may  be
     referenced in the laneconfig file as `$prefix'.

     The next 6 bytes (called the ESI) are used to uniquely iden-
     tify  a  host  system;  in  most  of the examples given, the
     system's hardware MAC address is used. The local MAC address
     may  be  referenced  in  the  laneconfig file as `$mac'. The
     final byte is a selector byte that may be used by  the  host
     for internal routing of data. Use of the predefined variable
     `sel' will guarantee that an appropriate value for the given
     interface will be used.

     Depending on the entry type,  as  determined  by  the  flags
     field,  some or all of the fields are required.  All entries
     must have an interface and flags field; the MAC Address/ELAN
     Name,  ATM Address, and VC field vary depending on the entry
     type.  The following sections describe the use of  variables
     in the laneconfig file, and the flag types, listing which of
     the MAC Address/ELAN Name, ATM Address, and  VC  fields  are
     required  for  that  type.  In  all  entries, an empty field
     should be indicated by a hyphen `-'.

OPTIONS
  Variables
     Because the prefix portion of an ATM address  specifies  the
     ATM  switch,  a  number  of hosts specified in an laneconfig
     file may have ATM addresses who share the  same  prefix.  To
     simplify  setting  up  the  laneconfig  file, one can define
     variables that contain part of an ATM address. A  variable's
     name  is an identifier consisting of a collection of no more
     than 32 letters, digits, and underscores  (`_').  The  value
     associated  with  the  variable  is denoted by a dollar sign
     (`$') followed immediately by the variable name.

     Variables may only be  used  in  the  ATM  and  MAC  address
     fields.   They may not be used in any of the other fields in
     an entry.

     Multiple variables may be concatenated to represent a single
     ATM  address expression. A colon must be used to concatenate
```

CODE EXAMPLE 3-7 laneconfig(4) Man Page *(Continued)*

```
     the variables.  Thus, if one variable, v1, is set to `11:22'
     and  another,  v2,  is  set to `33:44', the sequence $v1:$v2
     represents `11:22:33:44'. Hexadecimal numbers  may  also  be
     included  with  variables  in the expression. The expression
     `45:$v1:$v2' would have the value `45:11:22:33:44'.

     Variables are defined in the laneconfig  file  according  to
     the following format:

          set VARIABLE = EXPRESSION

     where VARIABLE is the name of a variable and  EXPRESSION  is
     an  expression  concatenating  one- or two-digit hexadecimal
     numbers and/or the values of variables that have been previ-
     ously defined.  The equal sign is optional, but the variable
     and  expression  must  be  separated  by  either  whitespace
     (spaces or tabs), an equal sign, or both.

     Several predefined variables are  built  in  to  the  SunATM
     software.  They include:

     prefix        the 13-byte prefix associated  with  the  local
                   switch.

     mac           the 6-byte  MAC  address  associated  with  the
                   local host or interface.

     sel           the  default  1-byte  Selector  for  the  local
                   interface.

     macsel        the concatenation of $mac:$sel.

     myaddress     the concatenation of $prefix:$mac:$sel, result-
                   ing in the default address for the local inter-
                   face.

     In most network configurations, the ATM address assigned  to
     the  local  interface will be myaddress; using this variable
     in the `l' entry makes it possible to use identical lanecon-
     fig  files  on all LAN Emulation clients in a given ATM net-
     work.

     Several rules apply to the use of variables in the  lanecon-
     fig file:

          Two variables cannot follow each other in an expression
```

CODE EXAMPLE 3-7 `laneconfig(4)` Man Page *(Continued)*

```
            without  an  intervening  colon. Thus, $v1:$v2 is legal
            whereas $v1$v2 is not.

            Fields  in  each  line  in  the  laneconfig  file   are
            separated  by  whitespace.   Therefore variables should
            not be separated from the rest of an ATM  address  with
            whitespace. For example, $v1: $v2 is illegal.

            Once a variable is defined by a set command, it may not
            be redefined later in the laneconfig file.

            The reserved  variable  names  may  not  be  set.  They
            include  `prefix',  `mac',  `sel', `macsel', and `myad-
            dress'.

    Basic Configuration Flags
       l    This flag identifies an entry for a local interface  on
            a LAN Emulation client.

            The ATM address is required.  The  MAC  address  should
            not  be  entered;  the  system will use the MAC address
            assigned to this physical interface.  No VC  should  be
            entered  either, since there will typically be multiple
            VCs over the local interface.

       t    Adds this MAC-ATM address or MAC address-VC pair to the
            local table.

            The MAC address is required; either an ATM address or a
            VC  field  is required, depending on whether a SVC or a
            PVC connection is desired.  If a mixture of SVC and PVC
            connections  is  desired,  both an ATM address and a VC
            are allowed.

       n    Specifies the name of the Emulated LAN.  Most LAN  Emu-
            lation  Services  will fill the Emulated LAN name in in
            configuration and  join  requests  from  LAN  Emulation
            Clients,  but  this is not always the case. If your LAN
            Emulation Services do not provide  Emulated  LAN  names
            for  client  requests,  you can include the name in the
            laneconfig file.

            The Emulated LAN name is required; the ATM address  and
            VC fields are illegal.

       The required, optional, and illegal  fields  for  the  basic
```

CODE EXAMPLE 3-7 laneconfig(4) Man Page *(Continued)*

```
      flag types are summarized in the following table:

      --------------------------------------------------------
      Interface   MAC-Addr/       ATM-Addr     VCI      FLAGS
                  ELAN Name
      --------------------------------------------------------
      required    illegal         required     illegal    l
      required    MAC-Addr req.   xor          xor*       t
      required    ELAN-Name req.  illegal      illegal    n
      --------------------------------------------------------
      * one or the other is required; both are illegal.

   Advanced Configuration Flags
      The basic configuration flags are sufficient for most  stan-
      dard  network  configurations.   However, since networks are
      rarely homogeneous, there may be cases in which, for intero-
      perability  purposes, a network must be configured with dif-
      ferent characteristics than the defaults that are built into
      the  SunATM adapter, or with unusual addressing schemes that
      require more than the basic  configuration  flags  described
      above.  The following flags may also be used in the lanecon-
      fig file to alter the default behavior when necessary.

      c    Specifies an alternate LECS address.  By  default,  the
           SunATM  software  uses ILMI to query the switch for the
           LECS address, then falls back to the well-known address
           if  ILMI  is not available or if the switch cannot pro-
           vide the LECS address via ILMI.  If, however, you  wish
           to specify an alternate LECS, or you wish to connect to
           the LECS over a PVC, you may provide the alternate  ATM
           address or VCI in this entry. If you wish to make a PVC
           connection, the VCI must be 17, as required by the  LAN
           Emulation standard.

           Either an ATM address or a VC field  must  appear;  the
           MAC address should not appear.

      s    Specifies the LES address or  VCI,  and  instructs  the
           system  to contact the LES directly, and to use default
           subnet configuration information. This flag  should  be
           used  if  your  ATM  network  does not have an LECS. By
           default (no `s' entry), the system  first  connects  to
           the LECS, which provides the LES address and configura-
           tion information.

           Either the ATM address or a VC is  required.   The  MAC
```

CODE EXAMPLE 3-7 laneconfig(4) Man Page *(Continued)*

```
          address should not appear.

     a    Specifies an address that may have access to this host.
          If no `a' entries appear in the laneconfig file, access
          to the host  is  unrestricted.  Including  `a'  entries
          allows  access  to  be restricted to known hosts. As an
          alternative to listing individual  addresses,  the  ATM
          address  field  may  contain  a prefix, followed by the
          wildcard   $anymacsel,   which   matches   any   7-byte
          ESI/Selector  combination  following  the given prefix.
          This allows access by any host connected to the  switch
          specified  by the given prefix.  NOTE: If this value is
          changed, only a reboot will ensure that  old  addresses
          are not being cached.

          An ATM address is required; neither the MAC address nor
          the VCI should appear.

     m    Specifies manual address configuration mode. This indi-
          cates  to the system that ILMI is not being used on the
          specified interface. Entries  for  non-ILMI  interfaces
          may  not  use  the $prefix variable, or variables which
          make  use  of   $prefix   (such   as   $myaddress   and
          $localswitch_server),  since  ilmid will not be able to
          provide this information.

          Only the interface is required. The  MAC  address,  ATM
          address, and VCI should not appear.

     M    Specifies a larger MTU size.  By default, the LAN  Emu-
          lation software will be configured for a 1516-byte MTU.
          If a larger size is supported by and configured on your
          LAN  Emulation  services,  it may be set in this entry.
          The valid values are 1516 (1500 bytes of data, 16 bytes
          of  LANE  header),  4544 (4528 bytes of data), and 9234
          (9218 bytes of data).

          The interface is required,  and  the  MTU  size  should
          appear  in  the  second field.  The ATM address and VCI
          should not appear.

     The required, optional, and illegal fields for the  advanced
     flag types are summarized in the following table:

     --------------------------------------------------------
     Interface   MAC-Addr/       ATM-Addr     VCI      FLAGS
```

CODE EXAMPLE 3-7 laneconfig(4) Man Page *(Continued)*

```
                 ELAN Name
     --------------------------------------------------------
     required    illegal         xor          xor*       c
     required    illegal         xor          xor*       s
     required    illegal         required     illegal    a
     required    illegal         illegal      illegal    m
     required    MTU size        illegal      illegal    M
     --------------------------------------------------------
     * one or the other is required; both are illegal.

EXAMPLES
     The following example shows a basic LAN  Emulation  Client's
     laneconfig  file. The local information is provided, as well
     as the addresses of a frequently used  server.  The  use  of
     variables is also demonstrated.


          set srvr_mac = 08:00:20:01:02:03

          ba0 -          $myaddress          -    l
          ba0 $srvr_mac  $prefix:$srvr_mac   -    t

     The following example shows the laneconfig file  for  a  LAN
     Emulation Client whose LECS requires that the client include
     the Emulated LAN name in its messages.


          ba1 -          $myaddress     -    l
          ba1 elan1      -              -    n

     The following example shows the laneconfig file  for  a  LAN
     Emulation Client whose ATM network does not include an LECS.


          set les_mac = 01:02:03:04:05:06

          ba0 -          $myaddress          -    l
          ba0  -         $prefix:$les_mac    -    s

SEE ALSO
     lanesetup(1M)

     ATM Forum, LAN Emulation Over ATM Specification Version 1.0,
     LAN Emulation SWG Drafting Group.

NOTES
```

CODE EXAMPLE 3-7 `laneconfig(4)` Man Page *(Continued)*

```
     Each entry should be entered on one line with no  breaks  or
     carriage returns.
```

mib.rt(4)

CODE EXAMPLE 3-8 mib.rt(4) Man Page

```
mib.rt(4)                 File Formats                 mib.rt(4)



NAME
     mib.rt - SunATM SNMP agent utility file.

SYNOPSIS
     /etc/opt/SUNWatm/snmp/mib.rt

DESCRIPTION
     The mib.rt file contains a  listing  of  object  identifiers
     used  by  atmsnmpd(1M) to translate the symbolic names found
     in acl.cfg(4), context.cfg(4), party.cfg(4) and view.cfg(4).

     Each line is of the form:

          $obj <object-identifier> <descriptor>

     where:

     <object-identifier> is a sequence of  non-negative  integers
     separated  by  dots, and identifies the OBJECT IDENTIFIER of
     the symbolic name used in the configuration files  mentioned
     above.

     <descriptor> is the symbolic name associated with the OBJECT
     IDENTIFIER.

     For example:

          $obj      1.3.6.1.6      snmpV2

     Whenever you want to use some descriptor in  the  configura-
     tion files that is not defined in the mib.rt file, you could
     extend this file to contain it. All  the  ancestors  of  the
     name  that  you  are defining must be specified as well. For
     example, in order to add  internet  (1.3.6)  you  must  also
     define dod (1.3) and iso (1).
```

CODE EXAMPLE 3-8 `mib.rt(4)` Man Page *(Continued)*

```
     atmsnmpd(1M) builds a representation of this file in  memory
     when  it  is first started, so additions to this file or any
     of the configuration  files  will  not  take  effect  unless
     atmsnmpd(1M) is restarted.

SEE ALSO
     atmsnmpd(1M),  acl.cfg(4),   context.cfg(4),   party.cfg(4),
     view.cfg(4)
```

party.cfg(4)

CODE EXAMPLE 3-9 party.cfg(4) Man Page

```
party.cfg(4)              File Formats               party.cfg(4)



NAME
     party.cfg - SunATM SNMP party database  group  configuration
     file

SYNOPSIS
     /etc/opt/SUNWatm/snmp/party.cfg

DESCRIPTION
     The party.cfg file  contains  the  party  database  for  the
     SunATM  SNMP  agent,  amtsnmpd(1M). The entries contained in
     this file are the conceptual rows  of  the  partyTable  (RFC
     1447).

     Each conceptual row contains the following entries:

     partyIdentity        A  unique  object  identifier  for  the
                          party.

     partyTDomain         Indicates transport  service  by  which
                          the  party  receives network management
                          traffic.

     partyTAddr           The transport service  address  of  the
                          party.  For  snmpUDPDomain, the address
                          is formatted as a  4-octet  IP  address
                          concatenated  with  a  2-octet UDP port
                          number.

     partyMaxMessageSize  An integer in the range 484  to  65,507
                          that  represents  the  maximum  message
                          length in octets that this  party  will
                          accept.

     partyLocal           An indication of whether this party  is
                          local to the agent.
```

CODE EXAMPLE 3-9 party.cfg(4) Man Page *(Continued)*

```
     partyAuthProtocol   Object identifier of the authentication
                         protocol, if any.

     partyAuthPrivate    An  encoding  of  the  party's  private
                         authentication key, or value, needed to
                         support the authentication prorocol.

     partyAuthLifetime   A  non-negative  integer  which  is  an
                         upper bound of the lifetime of the mes-
                         sage in seconds.

     partyPrivProtocol   Object identifier of the privacy proto-
                         col, if any.

     partyPrivPrivate    An  encoding  of  the  party's  private
                         encryption  key,  needed to support the
                         privacy protocol.

     partyStorageType    The storage type  for  this  conceptual
                         row  in  the  partyTable.  Takes on the
                         values 1-4.

     partyStatus         The status of this  conceptual  row  in
                         the  partyTable.  Takes  on  the values
                         valid (1) and invalid (2).

     partyAuthClock      The    authentication    clock    which
                         represents  the  local  notion  of  the
                         current time  specific  to  the  party.
                         This  value  must  not  be  decremented
                         unless the party's private  authentica-
                         tion key is changed simultaneously.

     authTimestamp       Represents the time of  the  generation
                         of  the  message  according to the par-
                         tyAuthClock of the SNMP party that ori-
                         ginated  it.  The  granularity  of  the
                         clock, and therefore of this timestamp,
                         is 1 second (RFC 1352).

     Each entry in the file is represented by 10 lines.

           partyIdentity
           partyStatus
           partyLocal partyStorageType partyMaxMessageSize
           partyAuthLifetime authTimestamp partyAuthClock
```

CODE EXAMPLE 3-9 `party.cfg(4)` Man Page *(Continued)*

```
           partyTDomain
           partyTAddr
           partyAuthProtocol
           partyAuthPrivate (in hex)
           partyPrivProtocol
           partyPrivPrivate (in hex)

     Symbolic names may be used as long as  they  appear  in  the
     mib.rt(4)  file.  Otherwise  the  dotted  object ids must be
     used.  ';' is the comment character.  Comments may not be in
     between sections of a party.

EXAMPLES
     The following is an example of a typical party entry in  the
     party.cfg file.

           initialPartyId.127.0.0.1.1
           001
           001 003 01400
           000000000300 0000000000 0000000000
           snmpUDPDomain
           0000000000
           noAuth
           <empty line>
           noPriv
           <empty line>

     This entry defines the partyIdentity object identifier, par-
     tyStatus  active  (1), partyLocal true (1), partyStorageType
     nonVolatile (3), partyMaxMessageSize 1400 bytes,  partyAuth-
     Lifetime  300  seconds, authTimestamp zero (no authenticated
     message from the party has  been  received),  partyAuthClock
     zero,  partyAuthProtocol  noAuth  (no  authentication), par-
     tyPrivProtocol noPriv (the protocol without privacy), and no
     authentication  keys  (partyAuthPrivate and partyPrivPrivate
     are the empty strings).

SEE ALSO
     atmsnmpd(1M),   view.cfg(4),   acl.cfg(4),   context.cfg(4),
     mib.rt(4)
```

view.cfg(4)

CODE EXAMPLE 3-10 view.cfg(4) Man Page

```
view.cfg(4)               File Formats                view.cfg(4)



NAME
     view.cfg - SunATM SNMP MIB-view database group configuration
     file

SYNOPSIS
     /etc/opt/SUNWatm/snmp/view.cfg

DESCRIPTION
     The view.cfg file contains the  MIB-view  database  for  the
     SunATM  SNMP  agent,  amtsnmpd(1M). The entries contained in
     this file are the conceptual  rows  of  the  viewTable  (RFC
     1447).

     Each conceptual row contains the following entries:

     viewIndex        A unique value for each MIB view.

     viewSubtree      A MIB Subtree.

     viewMask         The bit mask which, in combination with the
                      corresponding   instance   of  viewSubtree,
                      defines a family of view subtrees.

     viewType         Takes on the values included (1),  excluded
                      (2).  Indicates  whether  the  coresponding
                      family of view subtrees defined by viewSub-
                      tree  and  viewMask is included or excluded
                      from the MIB view.

     viewStorageType  The storage type for this conceptual row in
                      the ViewTable. Takes on the values 1-4.

     viewStatus       The status of this conceptual  row  in  the
                      viewTable.  Takes  on  the values 1 (valid)
                      and 2 (invalid).
```

CODE EXAMPLE 3-10 `view.cfg(4)` Man Page *(Continued)*

```
     Each entry in the file is represented by 4 lines.

           viewIndex:viewSubtree
           viewStatus
           viewStorageType viewType
           viewMask

     Symbolic names may be used as long as  they  appear  in  the
     mib.rt(4)  file.  Otherwise  the  dotted  object ids must be
     used.  ';' is the comment character.  Comments may not be in
     between sections of a view.

EXAMPLES
     The following is an example of a typical view entry  in  the
     view.cfg file.

           1:dod
           001
           003 001
           <empty line>

     This entry defines  a  viewIndex  (1)  for  the  viewSubtree
     (dod),  viewStatus  active  (1), viewStorageType nonVolatile
     (3), viewType included (1) and no viewMask (empty line)

SEE ALSO
     atmsnmpd(1M),  acl.cfg(4),   party.cfg(4),   context.cfg(4),
     mib.rt(4)
```

CHAPTER 4

Special Files

The man pages in this chapter describe the various device and network interfaces available with the SunATM software.

TABLE 4-1 Special Files Man Pages

Man Page	Description	Page Number
`ba(7)`	SunATM device driver	page 112
`q93b(7)`	Multiplexing Driver supporting Q.2931 signalling	page 119

ba(7)

CODE EXAMPLE 4-1 ba(7) Man Page

```
ba(7)             Device and Network Interfaces             ba(7)



NAME
     ba - Sun ATM device driver

SYNOPSIS
     #include <sys/stropts.h>
     #include <atm/atm.h>
     #include <atm/atmioctl.h>

DESCRIPTION
     The ba driver is a Solaris 2.x  DDI/DKI  compliant  MT  safe
     STREAMS  device  driver. It presents a DLPI interface to the
     upper layers and supports M_DATA fastpath  and  M_DATA  raw.
     The   hardware  interface  supports  the  SunATM-155  Fiber,
     SunATM-155 UTP, and SunATM-622 products.

     The two modes of operation that should be used  by  applica-
     tion  programs  are  raw mode and dlpi mode.  The default is
     dlpi mode.  By sending down a DLIOCRAW ioctl the raw mode is
     requested.  The mode chosen defines the format in which data
     should be sent to the driver.

     Raw mode implies that the four-byte vpci will be sent in the
     first  mblk followed by data in the first and any subsequent
     mblks. When a message is received on a vpci running  in  raw
     mode, the four-byte vpci will be sent up with the data.

     DLPI mode implies that two or more mblocks will be  sent  to
     the  driver.   The first, of type M_PROTO, contains the dlpi
     message type, which  is  dl_unitdata_req  for  transmit  and
     dl_unitdata_ind  for  receive.  The vpci is included in this
     mblock as well. The dl_unitdata_req and dl_unitdata_ind header
    formats are deined in the
     header file <sys/dlpi.h>.  The second and subsequent mblocks
     are of type M_DATA and contain the message.  When the driver
     gets  the  two  mblocks from the upper layer, it will remove
     the first mblock, and transmit the message.  On receive,
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
     the M_PROTO mblock is added, and the two-mblock structure
     is sent up to the user.

    A method of encapsulation must also be chosen; the method of
    encapsulation is specified when the VC is associated with a stream
    (using the A_ADDVC ioctl).  Currently, null and LLC encapsulation
    are supported.  Null encapsulation implies that a message consists
    only of data preceded by a four-byte vpci.  This type of encapsulation
    is most commonly used with raw mode.  LLC encapsulation implies that
    an LLC header precedes the data.  This header will include the SAP
    associated with the application’s stream (using DL_BIND_REQ).  This
    type of encapsulation is typically used with dlpi mode traffic.

    For LLC-encapsulated traffic, the driver will automatically add the
    LLC header on transmit if the stream is running in dlpi mode.  The
    driver will also strip the LLC header from incoming traffic before
    sending it up a dlpi mode stream.  In raw mode, however, the driver
    does not modify the packets at all; this includes the LLC header.
    Thus, an application using raw mode and LLC encapsulation must include
    its own LLC headers on transmit and will receive data with the LLC
    header intact.

    Received packets are directed to application streams by the driver
    based on the type of encapsulation.  If a packet is null-encapsulated,
    it will be sent up the stream associated with the vpci on which the
    packet was received.  If a packet is LLC-encapsulated, it will be
    sent to the stream which has bound (using DL_BIND_REQ) the SAP found
    in the LLC header.

     The driver  supports  several  of  the  DLPI  message  types
     defined  in  the  <sys/dlpi.h>  header  file.  Specifically,
     users  of  the  ba  driver  may   use   the   DL_ATTACH_REQ,
     DL_DETACH_REQ,  DL_BIND_REQ, DL_UNBIND_REQ, DL_UNITDATA_IND,
     and DL_UNITDATA_REQ.  In addition, a Sun-specific dlpi ioctl
     is  supported, DLIOCRAW.  There is no data structure associ-
     ated with the DLIOCRAW ioctl; simply a strioctl struct  with
     ic_cmd  set  to  DLIOCRAW may be used to set a stream to raw
     mode.

     The driver also supports the ATM-specific  ioctls  described
     below.   Definitions  for  the ioctl commands and structures
     may be found in <atm/atmioctl.h>.

IOCTLS
     The driver supports a  set  of  ioctl  functions  which  are
     called  using  the I_STR ioctl and strioctl structure as the
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
     argument.   See   the   streamio(7)   man   page   and   the
     <sys/stropts.h>  header  file  for  more information on this
     type of ioctl call.

     The commands supported in the ic_cmd field of  the  strioctl
     structure  are  described  in the following paragraphs.  The
     structures that the ic_dp field should  point  to  are  also
     described for each command.

     A_ALLOCBW      Allocate constant bit rate bandwidth for this
                    stream.  ic_dp should point to an a_allocbw_t
                    structure, which is defined as:

                         typedef struct {
                              int bw;
                         } a_allocbw_t;

                    In  this  ioctl  the  bandwidth   amount   is
                    expressed  as  an  integer number of megabits
                    per second (Mbps). See the  table  below  for
                    the amount of bandwidth available to be allo-
                    cated by the user.  All unallocated bandwidth
                    is  given  to  IP  and dlpi mode traffic. The
                    A_ALLOCBW ioctl is supported for  compatibil-
                    ity  with  software  written  for SunATM 1.0.
                    The  A_ALLOCBW_CBR  ioctl  provides  a  finer
                    granularity in bandwidth allocation.

     A_ALLOCBW_CBR  Allocate constant bit rate bandwidth for this
                    stream.   ic_dp   should   point   to   an
                    a_allocbw_cbr_t structure, which  is  defined
                    as:

                         typedef struct {
                              int bw;
                         } a_allocbw_cbr_t;

                    In  this  ioctl  the  bandwidth   amount   is
                    expressed  as an integer number of 64 kilobit
                    per second (Kbps) units. See the table  below
                    for  the  amount of bandwidth available to be
                    allocated  by  the  user.   All   unallocated
                    bandwidth  is  given  to  IP  and  dlpi  mode
                    traffic.

     A_ALLOCBW_VBR  Allocate variable bit rate bandwidth for this
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
stream.    ic_dp    should    point   to   an
a_allocbw_vbr_t structure, which  is  defined
as:

     typedef struct {
          int peak_bw;
          int avg_bw;
          int max_burst;
          int priority;
     } a_allocbw_vbr_t;

A_ALLOCBW_VBR implements  the  GCRA  (Generic
Cell  Rate  Algorithm)  as defined by the ATM
Forum UNI 3.0 specification.   peak_bw  speci-
fies  (in  64 Kbps units) the Peak Cell Rate.
avg_bw specifies (in 64 Kbps units) the  Sus-
tainable  Cell  Rate. max_burst specifies the
number of cells which can  be  sent  back  to
back  on  the  media,  the Maximum Burst Size
from  the   UNI   spec.   priority   can   be
AVBR_HIGH_PRI  or  AVBR_LO_PRI. AVBR_HIGH_PRI
will always get  their  requested  bandwidth,
AVBR_LO_PRI can starve if other users request
all available bandwidth.

Note that the peak_bw, avg_bw, and  max_burst
parameters  are enforced by the hardware dev-
ice. Since the  hardware  is  not  infinitely
programmable  the  driver  may have to modify
the requested B/W before programming the dev-
ice.  The  driver  will  program the hardware
avg_bw as close to  the  requested  value  as
possible.  peak_bw  may  be  rounded  down as
necessary to meet the  hardware  granularity;
the received peak_bw will always be less than
or equal  to  the  requested  peak_bw,  never
greater.  max_burst  will be truncated at the
maximum  supported  by  the  hardware;   the
received  max_burst  will always be less than
or equal to the  requested  max_burst,  never
greater.

See the table below for the amount  of  (sus-
tained)  bandwidth  available to be allocated
by the user.  All  unallocated  bandwidth  is
given to IP and dlpi mode traffic.
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
Available Bandwidth
--------------------------------------------------------------
| Product              |    SunATM-155    |    SunATM-622    |
|----------------------+------------------+------------------|
| Unit of Measure      |  Mbps  | 64 Kbps |  Mbps  | 64 Kbps |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Total Bandwidth      |  155   |  2480   |  622   |  9952   |
|                      |        |         |        |         |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Cell Header/Phy      |   20   |   320   |   88   |  1408   |
|   Layer Overhead     |        |         |        |         |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Reserved by Software |  0.64  |     1   |  0.64  |     1   |
|                      |        |         |        |         |
|----------------------+--------+---------+--------+---------|
|                      |        |         |        |         |
| Available to User    |134.875 |  2158   |533.875 |  8542   |
|                      |        |         |        |         |
--------------------------------------------------------------


A_RELSEBW      Release bandwidth that was  previously  allo-
               cated  for this stream. ic_dp should point to
               an a_allocbw_t structure.


On successful completion, the ALLOCBW/RELSEBW ioctls  return
0 . Otherwise, -1 is returned and errno is set to one of the
following values:

          EUNATCH   The user has not attached to a ppa.

          EINVAL    The requested bandwidth is  negative  or
                    otherwise invalid.

          ENOSPC    All useable bandwidth has  already  been
                    allocated,  or  no  bandwidth  group  is
                    available.

          EDEADLK   (VBR only) The requested  peak  rate  is
                    less  than  the  requested average rate.
                    The traffic parameters are impossible to
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
                    satisfy.



A_ADDVC        Add a vpci to those serviced by this  stream,
               and  specify  the  encapsulation  type.   The
               encapsulation  type  defines  the  format  in
               which  data  will  be sent to the driver: raw
               mode, indicated by NULL_ENCAP, implies a sin-
               gle  mblock with only the four-byte vpci fol-
               lowed immediately by the  data.   dlpi  mode,
               indicated  by LLC_ENCAP, implies a two-mblock
               message, consisting of a M_PROTO mblock  fol-
               lowed by a M_DATA mblock containing the data.
               The M_PROTO mblock will contain a  dlpi  mes-
               sage       type      (dl_unitdata_req      or
               dl_unitdata_ind) and the vpci; the format may
               be  found  in  <sys/dlpi.h>.  For the A_ADDVC
               ioctl call,  ic_dp  points  to  an  a_addVC_t
               structure, which is defined as:

                    typedef struct {
                       vci_t vp_vc;   /* vpci to be added */
                       int    aal_type;/* null -> 0,       */
                                       /* AAL5 -> 5        */
                       int    encap;   /* encapsulation; see  */
                                       /* <atm/atmioctl.h> for */
                                       /* possible values     */
                       int    buf_type;/* if AAL5:             */
                                       /*  0 -> small buf (9 k) */
                                       /*  1 -> big buf (9 k)   */
                                       /*  2 -> huge buf (64 k) */
                                       /* if null AAL           */
                                       /*   -> # of cells       */
                    } a_addVC_t;

A_DELVC        Remove a vpci from  those  serviced  by  this
               stream.   ic_dp points to an a_delVC_t struc-
               ture:

                    typedef struct {
                         vci_t  vp_vc;
                    } a_delVC_t;
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
     On successful completion, the ADDVC/DELVC ioctls return 0  .
     Otherwise,   - 1  is returned and errno is set to one of the
     following values:


               EUNATCH   The user has not attached to a ppa.

               EINVAL    The encap argument  is  not  valid,  the
                         aal_type  is  not  valid, or the size is
                         too large. The  hardware  controlled  by
                         the  ba  driver supports frames up to 64
                         KBytes.

               E2BIG     The VCI is outside the  range  supported
                         by  the  hardware.   The  hardware  con-
                         trolled by the ba driver  supports  VCIs
                         0-1023.

               EBUSY     The requested VCI is in use  by  another
                         process.

               ENOMEM    Memory allocation failed. Resources  for
                         the  HUGE_BUF_TYPE  buffer  ring are not
                         allocated by the  driver  until  a  user
                         requests them.



EXAMPLES
     The following code fragment demonstrates opening a ba device
     and  allocating  128 Kbits/sec of bandwidth for that stream.
     The example shows the actual ioctl  to  set  the  bandwidth.
     There  is a utility function in libatm, atm_allocate_cbr_bw,
     to make this task easier.

          #include <sys/types.h>
          #include <stropts.h>
          #include <sys/conf.h>
          #include <atm/atm.h>
          #include <atm/atmioctl.h>

          int
          main (int argc, char **argv)
          {
              char              dev[0x20] = "/dev/ba0";
              int               fd;
```

CODE EXAMPLE 4-1 ba(7) Man Page *(Continued)*

```
              int               ppa = 0;
              a_allocbw_cbr_t   ap;
              struct strioctl   strioctl;

              if ((fd = atm_open(dev)) < 0) {
                  exit(-1);
              }

              if (atm_attach(fd, ppa, -1) < 0) {
                  exit(-1);
              }

              ap.bw = 2;

              strioctl.ic_cmd = A_ALLOCBW_CBR;
              strioctl.ic_timout = -1;
              strioctl.ic_len = sizeof (ap);
              strioctl.ic_dp = (caddr_t) &ap;

              if (ioctl(fd, I_STR, &strioctl) < 0) {
                  exit(-1);
              }
          }


SEE ALSO
     atm_util(3), dlpi(7), streamio(7)
```

q93b(7)

CODE EXAMPLE 4-2 q93b(7) Man Page

```
q93b(7)            Device and Network Interfaces          q93b(7)



NAME
     q93b - Multiplexing Driver supporting Q.2931 signalling

SYNOPSIS
```

CODE EXAMPLE 4-2 q93b(7) Man Page *(Continued)*

```
     #include <atm/qcc.h>
     #include <atm/qccioctl.h>

DESCRIPTION
     The q93b driver supports Q.2931 call control  signalling  as
     defined  by  the  ATM  Forum's User Network Interface, V3.0,
     V3.1, and V4.0.  It is a multi-threaded, loadable, clonable,
     M-to-N   multiplexing  STREAMS  driver.   Its  interface  is
     defined by the Q.2931 message set, with some  additions  for
     synchronization  between  the  driver  and  user process.  A
     Q.2931 Call Control library is provided  with  the  SUNWatma
     software  package which provides a set of functions that may
     be used to build and parse q93b messages.  See the qcc_* man
     pages for further information.

     The following table lists the messages types that  are  sup-
     ported.  For sample message exchanges, see Appendix E in the
     SunATM Manual.

          TYPE                 | DIRECTION
          --------------------------------
          setup                |  both
          setup_ack*           |  to user
          call_proceeding      |  both
          alerting             |  both**
          connect              |  both
          connect_ack          |  to user
          release              |  to q93b
          release_complete     |  both
          status_enquiry       |  to q93b
          status               |  to user
          notify               |  both**
          restart              |  both
          restart_ack          |  both
          add_party            |  to q93b
          add_party_ack        |  to user
          add_party_alerting   |  to user**
          add_party_reject     |  to user
          drop_party           |  both
          drop_party_ack       |  to q93b
          leaf_setup_fail      |  both**
          leaf_setup_req       |  both**

          *private to the user/q93b interface
          **only supported in UNI 4.0
```

CODE EXAMPLE 4-2 q93b(7) Man Page *(Continued)*

```
     Messages to the q93b driver should consist of two mblks,  as
     shown below:
                      M_PROTO                            M_DATA

                 _____________      ___________________________
           --->|             |--->|     |               |     |
               |  IF_Name    |    |  Q.2931 Message     |     |
               |  Call_ID    |    |     |               |     |
               |  Type       |    |     |               |     |
               |  Error      |    |     |  Information  |     |
               |  Call_Tag   |    | (9) |  Elements     | (16)|
               |_____________|    |_____|_______________|_____|

     The 9 byte header on the M_DATA block consists of the Q.2931
     header information; the 16 byte trailer is allocated for use
     by the lower layers to enhance performance.  This additional
     25 bytes is added to the variable length Information Element
     (IE)  section  when  the  qcc_len  functions  calculate  the
     required  buffer  sizes  for  the message types.  The Q.2931
     header is also filled in by the qcc_bld functions.

IOCTLS
     The q93b driver  supports  a  q93b-specific  STREAMS  ioctl,
     Q93B_IOC.   Several  commands  may  be  specified using this
     ioctl.  The data structure used varies depending on the com-
     mand;  see the <atm/qccioctl.h> header file for a definition
     of these structures.  Functional interfaces for these  ioctl
     commands  are  provided in the qcc library; see the qcc_util
     man page for descriptions of these functions.

     The following commands are supported:

     Q93B_IOC_BIND  Binds a stream to the q93b driver to a speci-
                    fied  service  access  point (sap).  The q93b
                    driver uses the sap, which must be  specified
                    in  the  BHLI  Information Element of a setup
                    message, to determine to which  of  its  user
                    streams  it  will send an incoming setup mes-
                    sage.

     Q93B_IOC_BIND_LIJID
                    Binds a stream to a specified  Leaf-Initiated
                    Join   ID.   Leaf-initiated  join  is  a  new
                    feature in UNI 4.0 signalling,  which  allows
                    an  endpoint  to  request  to  be  added to a
                    point-to-multipoint  connection.   The  leaf-
                    initiated  join id is used by the endpoint to
```

CODE EXAMPLE 4-2 q93b(7) Man Page *(Continued)*

```
                    identify the connection which  it  wishes  to
                    join.   In  order  to be the root of a point-
                    to-multipoint call which will  support  leaf-
                    initiated join, a user application must asso-
                    ciate its q93b stream with the leaf-initiated
                    join  id  in one of two ways: by setting up a
                    call in which the leaf-initiated join  id  is
                    specified,  or  by  sending this ioctl to the
                    q93b driver.

     Q93B_IOC_UNBIND_LIJID
                    Unbinds  a  Leaf-Initiated  Join  ID  from  a
                    stream.

SEE ALSO
     qcc_bld(3),    qcc_create(3),    qcc_len(3),    qcc_pack(3),
     qcc_parse(3),   qcc_unpack(3),   qcc_util(3),   atm_util(3),
     qcc_bld(9F),  qcc_create(9F),  qcc_pack(9F),  qcc_parse(9F),
     qcc_unpack(9F), ba(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.

     "ATM User-Network Interface Specification, V3.1," ATM Forum.

     "ATM User-Network Interface Specification, V4.0," ATM Forum.

     "Data Link Provider Interface Specification, Rev. 2.0.0," 20
     Aug 1991, UNIX International.

     SunATM Manual
```

CHAPTER 5

DDI and DKI Kernel Functions

The man pages in this chapter describe the kernel functions available for use by the SunATM device drivers.

TABLE 5-1 DDI and DKI Kernel Function Man Pages

Man Page	Description	Page Number
qcc_bld(9F)	Build Q.2931 messages, with these commands: qcc_bld_setup(9F), qcc_bld_alerting(9F), qcc_bld_call_proceeding(9F), qcc_bld_connect(9F), qcc_bld_release(9F), qcc_bld_release_complete(9F), qcc_bld_status(9F), qcc_bld_status_enquiry(9F), qcc_bld_notify(9F), qcc_bld_restart(9F), qcc_bld_restart_ack(9F), qcc_bld_add_party(9F), qcc_bld_add_party_ack(9F), qcc_bld_party_alerting(9F), qcc_bld_add_party_reject(9F), qcc_bld_drop_party(9F), qcc_bld_drop_party_ack(9F),	page 127

TABLE 5-1 DDI and DKI Kernel Function Man Pages *(Continued)*

Man Page	Description	Page Number
	`qcc_bld_leaf_setup_fail(9F)`, `qcc_bld_leaf_setup_req(9F)`	
`qcc_create(9F)`	Build Q.2931 messages, including: `qcc_create_setup(9F)`, `qcc_create_alerting(9F)`, `qcc_create_call_proceeding(9F)`, `qcc_create_connect(9F)`, `qcc_create_connect_ack(9F)`, `qcc_create_release(9F)`, `qcc_create_release_complete(9F)`, `qcc_create_status(9F)`, `qcc_create_status_enq(9F)`, `qcc_create_notify(9F)`, `qcc_create_restart(9F)`, `qcc_create_restart_ack(9F)`, `qcc_create_add_party(9F)`, `qcc_create_add_party_ack(9F)`, `qcc_create_party_alerting(9F)`, `qcc_create_add_party_reject(9F)`, `qcc_create_drop_party(9F)`, `qcc_create_drop_party_ack(9F)`, `qcc_create_leaf_setup_fail(9F)`, `qcc_create_leaf_setup_req(9F)`	page 135
`qcc_pack(9F)`	Encode Q.2931 message structure information and pack into streams buffers, with these commands: `qcc_pack_setup(9F)`, `qcc_pack_alerting(9F)`, `qcc_pack_call_proceeding(9F)`, `qcc_pack_connect(9F)`, `qcc_pack_connect_ack(9F)`, `qcc_pack_release(9F)`,	page 144

TABLE 5-1 DDI and DKI Kernel Function Man Pages *(Continued)*

Man Page	Description	Page Number
	qcc_pack_release_complete(9F), qcc_pack_status(9F), qcc_pack_status_enq(9F), qcc_pack_notify(9F), qcc_pack_restart(9F), qcc_pack_restart_ack(9F), qcc_pack_add_party(9F), qcc_pack_add_party_ack(9F), qcc_pack_party_alerting(9F), qcc_pack_add_party_reject(9F), qcc_pack_drop_party(9F), qcc_pack_drop_party_ack(9F), qcc_pack_leaf_setup_fail(9F), qcc_pack_leaf_setup_req(9F)	
qcc_parse(9F)	Parse Q.2931 messages, with these commands: qcc_parse_setup(9F), qcc_parse_alerting(9F), qcc_parse_call_proceeding(9F), qcc_parse_connect(9F), qcc_parse_release(9F), qcc_parse_release_complete(9F), qcc_parse_status_enquiry(9F), qcc_parse_notify(9F), qcc_parse_status(9F), qcc_parse_restar(9F), qcc_parse_restart_ack(9F), qcc_parse_add_party(9F), qcc_parse_add_party_ack(9F), qcc_parse_party_alerting(9F), qcc_parse_add_party_reject(9F),	page 148

TABLE 5-1 DDI and DKI Kernel Function Man Pages *(Continued)*

Man Page	Description	Page Number
	qcc_parse_drop_party(9F), qcc_parse_drop_party_ack(9F), qcc_parse_leaf_setup_fail(9F), qcc_parse_leaf_setup_req(9F)	
qcc_set_ie(9F)	Add or update Information Elements in a Q.2931 message structure	page 156
qcc_unpack(9F)	Decode Q.2931 messages and unpack into message structures with these commands: qcc_unpack_setup(9F), qcc_unpack_alerting(9F), qcc_unpack_call_proceeding(9F), qcc_unpack_connect(9F), qcc_unpack_connect_ack(9F), qcc_unpack_release(9F), qcc_unpack_release_complete(9F), qcc_unpack_status(9F), qcc_unpack_status_enq(9F), qcc_unpack_notify(9F), qcc_unpack_restart(9F), qcc_unpack_restart_ack(9F), qcc_unpack_add_party(9F), qcc_unpack_add_party_ack(9F), qcc_unpack_party_alerting(9F), qcc_unpack_add_party_reject(9F), qcc_unpack_drop_party(9F), qcc_unpack_drop_party_ack(9F), qcc_unpack_leaf_setup_fail(9F), qcc_unpack_leaf_setup_req(9F)	page 162

qcc_bld(9F)

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page

```
qcc_bld(9F)       Kernel Functions for Drivers        qcc_bld(9F)



NAME
     qcc_bld,          qcc_bld_setup,           qcc_bld_alerting,
     qcc_bld_call_proceeding,  qcc_bld_connect,  qcc_bld_release,
     qcc_bld_release_complete,                    qcc_bld_status,
     qcc_bld_status_enquiry,   qcc_bld_notify,   qcc_bld_restart,
     qcc_bld_restart_ack,                      qcc_bld_add_party,
     qcc_bld_add_party_ack,               qcc_bld_party_alerting,
     qcc_bld_add_party_reject,                qcc_bld_drop_party,
     qcc_bld_drop_party_ack,             qcc_bld_leaf_setup_fail,
     qcc_bld_leaf_setup_req - build Q.2931 messages

SYNOPSIS
     cc -DKERNEL -D_KERNEL [ flag ... ] file ...

     #include <atm/types.h>
     #include <atm/qcc.h>

     char _depends_on[] = "drv/qcc";

     mblk_t *qcc_bld_setup(char *ifname, int calltag, int vci,
               int forward_sdusize, int backward_sdusize,
               atm_addr_t *src_addrp, atm_addr_t *dst_addrp,
               int sap, int endpt_ref);

     mblk_t *qcc_bld_alerting(char *ifname, int callid, int vci,
               int endpt_ref);

     mblk_t *qcc_bld_call_proceeding(char *ifname, int callid,
               int vci, int endpt_ref);

     mblk_t *qcc_bld_connect(char *ifname, int callid, int vci,
               int forward_sdusize, int backward_sdusize,
               int endpt_ref);

     mblk_t *qcc_bld_release(char *ifname, int callid,
               int cause);
```

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page *(Continued)*

```
mblk_t *qcc_bld_release_complete(char *ifname, int callid,
          int cause);

mblk_t *qcc_bld_status_enquiry(char *ifname, int callid,
          int endpt_ref);

mblk_t *qcc_bld_status(char *ifname, int callid,
          int callstate, int cause, int endpt_ref,
          int endpt_state);

mblk_t *qcc_bld_notify(char *ifname, int callid,
          int contentlen, u_char *contentp, int endpt_ref);

mblk_t *qcc_bld_restart(char *ifname, int callid, int vci,
          int rstall);

mblk_t *qcc_bld_restart_ack(char *ifname, int callid,
          int vci, int rstall);

mblk_t *qcc_bld_add_party(char *ifname, int callid,
          int forward_sdusize, int backward_sdusize,
          atm_address_t *src_addrp,
          atm_address_t *dst_addrp, int sap, int endpt_ref);

mblk_t *qcc_bld_add_party_ack(char *ifname, int callid,
          int endpt_ref);

mblk_t *qcc_bld_party_alerting(char *ifname, int callid,
          int endpt_ref);

mblk_t *qcc_bld_add_party_reject(char *ifname, int callid,
          int cause, int endpt_ref);

mblk_t *qcc_bld_drop_party(char *ifname, int callid,
          int cause, int endpt_ref);

mblk_t *qcc_bld_drop_party_ack(char *ifname, int callid,
          int cause, int endpt_ref);

mblk_t *qcc_bld_leaf_setup_fail(char *ifname, int callid,
          int cause, atm_address_t *dst_addrp,
          int leaf_num);

mblk_t *qcc_bld_leaf_setup_req(char *ifname, int leaftag,
          atm_address_t *src_addrp,
```

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page *(Continued)*

```
               atm_address_t *dst_addrp, int lij_callid);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The -DKERNEL and -D_KERNEL flags must be included  to  indi-
     cate  that  the  application should run in kernel space, and
     the qcc driver must be loaded (this requirement is expressed
     in  the  code  using  the  "depends_on"  line  shown  in the
     synopsis).

DESCRIPTION
     These functions build the various messages that make up  the
     Q.2931  protocol  which  is used for ATM signalling.  A full
     description of the message format and use can  be  found  in
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0. The messages built will conform to  the  ver-
     sion  of  the  UNI  Specification which is configured on the
     indicated interface.  The functions may be used by processes
     which are running in kernel space.

     In general, no error checking is performed on the data  that
     is  passed in.  Whatever data is passed in will be placed in
     the message that is built  without  examination.   The  only
     exceptions  to  this  are mentioned in the function descrip-
     tions.

     Two mblk_t structures are allocated and linked  by  each  of
     the  functions  (their  format  is  shown  in  the following
     diagram).  The  pointer  that  is  returned  points  to  the
     M_PROTO  block,  and  may then be passed downstream with the
     putq(9F) command.

                      M_PROTO                           M_DATA
                _____________      ___________________________
           --->|             |--->|     |               |     |
               |  IF_Name    |    |  Q.2931 Message     |     |
               |  Call_ID    |    |     |               |     |
               |  Type       |    |     |               |     |
               |  Error      |    |     |  Information  |     |
               |  Call_Tag   |    | (9) |  Elements     | (16)|
               |_____________|    |_____|_______________|_____|
```

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page *(Continued)*

```
     The parameters passed in to each function are used  to  fill
     in the data portions of these two mblks.

     Each function requires a minimum of  2  parameters:  ifname,
     which is a string containing the physical interface (such as
     ba0); and an integer, either calltag or callid, depending on
     the  message  type.   calltag  is  used in the setup message
     only; it is a reference number that is assigned by the  cal-
     ling  application.  callid is used in all other messages; it
     is assigned by the lower layer and will be sent  up  to  the
     user, with the calltag, in the setup_ack message.

     Other parameters for each function depend  on  the  type  of
     information  required for each message type, and are defined
     in the paragraphs describing each function call.

     qcc_bld_setup() constructs a setup  message  containing  the
     following  Information  Elements:  AAL  parameters, ATM user
     cell rate, broadband bearer capability, called party number,
     calling party number, quality of service parameter, and end-
     point reference.  The user must  pass  in  the  forward  and
     backward  sdu sizes for the AAL parameter IE, an ATM address
     for the destination for the called party number IE, and  one
     for  itself  for  the calling party number IE (atm_address_t
     format is defined in  the  <atm/qcc.h>  header  file).   The
     value  passed  in the sap parameter is placed in a broadband
     higher layer IE. The higher layer IE indicates  the  sap  to
     which  received  messages  should  be  directed. If the user
     passes in a positive vci, a connection identifier IE will be
     included;  if the user passes in a non-negative endpt_ref (0
     is valid), an endpoint reference IE will  be  included.  The
     endpoint  reference  IE  indicates  that this is a point-to-
     multipoint call.

     qcc_bld_alerting() is specific to UNI  4.0.   It  builds  an
     alerting message containing a  connection identifier IE if a
     positive vci is passed in, and an endpoint reference IE if a
     non-negative  endpt_ref is passed in.  An endpoint reference
     IE should only appear if the call is  a  point-to-multipoint
     call.  The alerting message is only supported under UNI 4.0.

     qcc_bld_call_proceeding() includes a  connection  identifier
     IE if a positive vci is passed in, and an endpoint reference
     IE if a non-negative endpt_ref is  passed  in.  An  endpoint
     reference  IE  should only appear if the call is a point-to-
     multipoint call.
```

CODE EXAMPLE 5-1 `qcc_bld(9F)` Man Page *(Continued)*

```
qcc_bld_connect() includes an AAL parameters  IE,  requiring
the forward_ and backward_sdusize values, a connection iden-
tifier IE if a positive vci value is passed in, and an  end-
point  reference  IE  if  a  non-negative endpt_ref value is
passed in. An endpoint reference IE should  only  appear  if
the call is a point-to-multipoint call.

qcc_bld_release() includes a cause IE  for  which  the  user
must  pass  in  a  cause  value.  The possible values can be
found in the <atm/qcc.h> header file.  The same is true  for
qcc_bld_release_ complete().

qcc_bld_status_enquiry() includes only an endpoint reference
IE  if  a non-negative endpt_ref value is passed in. An end-
point reference IE should only  appear  if  the  call  is  a
point-to-multipoint call.

qcc_bld_status() includes a call  state  IE,  requiring  the
user pass in the callstate parameter; possible values can be
found in the <atm/qcc.h> header file.  It  also  includes  a
cause  IE; the cause value must also be passed in.  Its pos-
sible values may also be found  in  the  <atm/qcc.h>  header
file.  Finally,  if  the call is a point-to-multipoint call,
endpoint reference  and  endpoint  state  IEs  may  also  be
included;  they  are  included  if  a non-negative endpt_ref
value is passed in. The endpt_state parameter is used in the
enpoint  state  IE; possible party state values may be found
in <atm/qcc.h>.

qcc_bld_notify() is specific to UNI 4.0.  It builds a notify
message,  including  a notification indicator IE, which con-
tains a buffer of user-defined information up to  a  maximum
length of 16 bytes (defined by contentlen and contentp), and
an endpoint reference IE if a non-negative  endpt_ref  value
is  passed  in.  An endpoint reference IE should only appear

if the call is a point-to-multipoint call.  The notify  mes-
sage is only valid under UNI 4.0.

qcc_bld_restart() includes a restart indicator IE, which  is
used to determine whether an individual call or all calls on
an interface should be restarted.  If rstall is 0, only  the
call  identified by vci should be restarted; in this case, a
connection identifier IE will also be included.   If  rstall
is  non-zero,  all calls will be restarted.  The same format
```

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page *(Continued)*

```
     applies to the qcc_bld_restart_ack() function.

     qcc_bld_add_party() constructs an add party  message  for  a
     point-to-multipoint  call. The message constructed will con-
     tain an AAL parameters IE, which includes the  forward_  and
     backward_sdusize  parameters,  a  calling  party  number IE,
     which includes the value pointed to by src_addrp,  a  called
     party  number  IE,  which  includes  the value pointed to by
     dst_addrp, a broadband  higher  layer  interface  IE,  which
     includes  the  sap  parameter, and an endpoint reference IE,
     which includes the endpt_ref parameter. The sap value in the
     broadband  higher  layer  information IE is used to indicate
     the sap to which the message should be passed by the receiv-
     ing host.

     qcc_bld_add_party_ack() constructs an add party ack  message
     which  includes  an  endpoint  reference  IE,  for which the
     endpt_ref parameter is required.

     qcc_bld_party_alerting() is specific to UNI 4.0.  It  builds
     a  party  alerting message, containing an endpoint reference
     IE, for which the endpt_ref parameter is required.

     qcc_bld_add_party_reject() includes a cause  IE,  containing
     the  cause value passed in. The possible cause values may be
     found in the <atm/qcc.h> header file. An endpoint  reference
     IE is also included, which requires the endpt_ref parameter.

     qcc_bld_drop_party() constructs a drop  party  message.  The
     message  constructed will contain two IEs: a cause IE, which
     requires the cause parameter, and an endpoint reference  IE,
     which  requires  the  endpt_ref  parameter.  Possible  cause
     values may be found in the header file <atm/qcc.h>.

     qcc_bld_drop_party_ack() contains an endpoint reference  IE,
     requiring  the  endpt_ref parameter, and optionally, a cause
     IE. The cause IE will be included if  a  positive  value  is
     passed  in in the cause parameter. Possible cause values may
     be found in the <atm/qcc.h> header file.

     qcc_bld_leaf_setup_fail() is specific to UNI 4.0.   It  con-
     tains a cause IE if a non-negative cause value is passed in;
     a called number IE if a non-null dst_addrp is passed in; and
     a  leaf  number  IE,  for  which  the  leaf_num parameter is
     required.  This message type is only valid under UNI 4.0.
```

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page *(Continued)*

```
     qcc_bld_leaf_setup_req() is specific to UNI  4.0.   It  con-
     tains  Calling  Number  and  Called  Number  IEs if non-null
     src_addrp and dst_addrp are passed in, respectively; it also
     contains  a leaf initiated join call identifier IE for which
     lij_callid is required, and a  leaf  number  IE.   The  leaf
     number  is  assigned  by  the q93b driver.  Because the leaf
     number is assigned by the q93b driver, a  mechanism  similar
     to  that  used  in  the setup and setup_ack messages is used
     with the leaf number: the  user  must  provide  a  'leaftag'
     parameter  in  the call to qcc_bld_leaf_setup_req();this tag
     is inserted in the calltag field of the  qcc  header.   When
     the  message  is received and accepted by the q93b driver, a
     leaf_setup_ack message  is  returned,  containing  both  the
     leaftag,  in  the  calltag  field of the qcc header, and the
     driver-assigned leaf number,  in  the  callref  field.   The
     leaf_setup_req and leaf_setup_ack messages are the only mes-
     sages which will not contain a call reference value  in  the
     callref  field; this is because the messages are not tied to
     a specific call.  This message, and the leaf-initiated  join
     functionality, are only supported under UNI 4.0.

RETURN VALUES
     All functions return a pointer to an mblk_t.  If  the  func-
     tion is not successful, the pointer will be NULL.

EXAMPLES
     The following code fragment builds a setup message and sends
     it downstream.

          #include <sys/stream.h>
          #include <atm/qcc.h>
          #include <atm/limits.h>

          char    _depends_on[] = "drv/qcc";

          void
          send_setup(queue_t *q);
          {
               mblk_t  *mp;
               char    ifname[QCC_MAX_IFNAME_LEN] = "ba0";
               int     calltag = 0x1234;
               int     vci = 0x100;
               int     forward_sdusize = 0x2378;
               int     backward_sdusize = 0x2378;
               int     sap = 0x100;
```

CODE EXAMPLE 5-1 qcc_bld(9F) Man Page *(Continued)*

```
               atm_addr_t     src_addr = {
                    0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
                    0x08, 0x00, 0x20, 0x1a, 0xe1, 0x53, 0x00
               };

               atm_addr_t     dst_addr = {
                    0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
                    0x08, 0x00, 0x20, 0x1a, 0xb6, 0xb9, 0x00
               };

               mp = qcc_bld_setup(ifname, calltag, vci,
                                   forward_sdusize, backward_sdusize,
                                   &src_addr, &dst_addr, sap, -1);

               if (putq(q, mp) < 0) {
                    perror("putq");
                    exit (-1);
               }
          }

SEE ALSO
     qcc_util(3), qcc_parse(9F), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.  These message types, if sent on an interface
     configured for UNI 3.0 or 3.1, will be discarded by the q93b
     driver  and  will  not  be sent out to the network.  The UNI
     4.0-specific messages are Alerting, Notify, Party  Alerting,
     Leaf  Setup Fail, and Leaf Setup Request, and are identified
     in the applicable function descriptions.
```

qcc_create(9F)

CODE EXAMPLE 5-2 qcc_create(9F) Man Page

```
qcc_create(9F)    Kernel Functions for Drivers     qcc_create(9F)



NAME
     qcc_create,      qcc_create_setup,      qcc_create_alerting,
     qcc_create_call_proceeding,              qcc_create_connect,
     qcc_create_connect_ack,                  qcc_create_release,
     qcc_create_release_complete,              qcc_create_status,
     qcc_create_status_enq,                    qcc_create_notify,
     qcc_create_restart,                  qcc_create_restart_ack,
     qcc_create_add_party,              qcc_create_add_party_ack,
     qcc_create_party_alerting,      qcc_create_add_party_reject,
     qcc_create_drop_party,            qcc_create_drop_party_ack,
     qcc_create_leaf_setup_fail,    qcc_create_leaf_setup_req   -
     create Q.2931 message structures

SYNOPSIS
     cc -DKERNEL -D_KERNEL [ flag ... ] file ...

     #include <atm/qcc.h>
     #include <atm/qcctypes.h>

     char _depends_on[] = "drv/qcc";

     int qcc_create_setup(qcc_setup_t *msgp, char *ifname,
           int calltag, atm_address_t *dst_addrp);

     int qcc_create_alerting(qcc_alerting_t *msgp, char *ifname,
           int callid);

     int qcc_create_call_proceeding(qcc_call_proc_t *msgp,
           char *ifname, int callid);

     int qcc_create_connect(qcc_connect_t *msgp, char *ifname,
           int callid);

     int qcc_create_connect_ack(qcc_connect_ack_t *msgp,
           char *ifname, int callid);
```

CODE EXAMPLE 5-2 `qcc_create(9F)` Man Page *(Continued)*

```
   int qcc_create_release(qcc_release_t *msgp, char *ifname,
      int callid, int cause);

   int qcc_create_release_complete(qcc_release_complete_t *
      msgp, char *ifname, int callid);

   int qcc_create_status_enq(qcc_status_enq_t *msgp,
      char *ifname, int callid);

   int qcc_create_status(qcc_status_t *msgp, char *ifname,
      int callid, int callstate, int cause);

   int qcc_create_notify(qcc_notify_t *msgp, char *ifname,
      int callid, int contentlen, u_char *contentp);

   int qcc_create_restart(qcc_restart_t *msgp, char *ifname,
      int callid, int indicator, int vci);

   int qcc_create_restart_ack(qcc_restart_ack_t *msgp,
      char *ifname, int callid, int indicator, int vci);

   int qcc_create_add_party(qcc_add_party_t *msgp,
      char *ifname, int callid, atm_address_t *dst_addrp,
      int endpt_ref);

   int qcc_create_add_party_ack(qcc_add_party_ack_t *msgp,
      char *ifname, int callid, int endpt_ref);

   int qcc_create_party_alerting(qcc_party_alerting_t *msgp,
      char *ifname, int callid, int endpt_ref);

   int qcc_create_add_party_reject(qcc_add_party_reject_t *
      msgp, char *ifname, int callid, int cause,
      int endpt_ref);

   int qcc_create_drop_party(qcc_drop_party_t *msgp,
      char *ifname, int callid, int cause, int endpt_ref);

   int qcc_create_drop_party_ack(qcc_drop_party_ack_t *msgp,
      char *ifname, int callid, int endpt_ref);

   int qcc_create_leaf_setup_fail(qcc_leaf_setup_fail_t *msgp,
      char *ifname, int callid, int cause,
      atm_address_t *dst_addrp, int leaf_num);

   int qcc_create_leaf_setup_req(qcc_leaf_setup_req_t *msgp,
```

CODE EXAMPLE 5-2 qcc_create(9F) Man Page *(Continued)*

```
           char *ifname, int leaftag, atm_address_t *src_addrp,
           atm_address_t *dst_addrp, int lij_callid);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The -DKERNEL and -D_KERNEL flags must be included  to  indi-
     cate  that  the  application should run in kernel space, and
     the qcc driver must be loaded (this requirement is expressed
     in  the  code  using  the  "depends_on"  line  shown  in the
     synopsis).

DESCRIPTION
     These functions create message structures  representing  the
     various  messages that make up the Q.2931 protocol, which is
     used for ATM signalling.  A full description of the  message
     format  and use can be found in the ATM Forum's User Network
     Interface Specification, V3.0, V3.1, or V4.0. The content of
     the  created  message structures will conform to the version
     of the UNI Specification which is configured  on  the  indi-
     cated  interface.  The  functions  may  be used by processes
     which are running in kernel space.

     After a message  structure  has  been  created,  non-default
     Information  Elements (IEs) may be added or existing IEs may
     be changed using the qcc_set_ie(9F) function. When the  mes-
     sage   structure   has   been   completely   specified,  the
     corresponding qcc_pack(9F)  function  should  be  called  to
     translate  the  message  structure  into the correct encoded
     format, contained in message  blocks  which  may  be  passed
     downstream using the putq(9F) function.

     In general, no error checking is performed on the data  that
     is  passed in.  Whatever data is passed in will be placed in
     the message that is built  without  examination.   The  only
     exceptions  to  this  are mentioned in the function descrip-
     tions.

     Each function requires a  minimum  of  3  parameters:  msgp,
     which  is  a  pointer  to  the appropriate message structure
     type; ifname, which is  a  string  containing  the  physical
     interface  (such  as ba0); and an integer, either calltag or
     callid, depending on the message type.  calltag is  used  in
```

CODE EXAMPLE 5-2 qcc_create(9F) Man Page *(Continued)*

```
     the  setup  message  only;  it is a reference number that is
     assigned by the calling application.  callid is used in  all
     other  messages;  it is assigned by the lower layer and will
     be sent up to the user, with the calltag, in  the  setup_ack
     message.

     The structure to which msgp points must be allocated by  the
     calling  user.  There is a unique structure for each message
     type;   the    message    structures    are    defined    in
     <atm/qcctypes.h>.

     Only the mandatory IEs for each message type  are  added  to
     the  message  structure  by  the qcc_create call.  The addi-
     tional parameters to the qcc_create functions allow the user
     to  define most of the information contained in those manda-
     tory IEs; however, in some cases default values are assumed.
     Those  values, as well as the additional parameters for each
     function, are indicated in the following paragraphs describ-
     ing each function call.

     qcc_create_setup() creates a setup  message  structure  con-
     taining  the  following  Information  Elements:  ATM traffic
     descriptor (called ATM cell  rate  in  UNI  3.0),  broadband
     bearer  capability, called party number, and quality of ser-
     vice parameter. The user must pass in  the  destination  ATM
     address for the called party number IE (atm_address_t format
     is defined in the <atm/types.h> header file).  The following
     default  values  are used for the remaining Information Ele-
     ments:

          ATM Traffic Descriptor:
               best effort; line rate is used for the forward and
               backward peak rates

          Broadband Bearer Capability:
               Bearer Class X, no indication for traffic type and
               timing  requirements, not susceptible to clipping,
               and point-to-point user plane

          Called Party Number:
               ATM Endsystem (NSAP) address type

          Quality of Service:
               Forward and backward class unspecified

     qcc_create_alerting() creates the structure for an  alerting
```

CODE EXAMPLE 5-2 `qcc_create(9F)` Man Page *(Continued)*

```
     message, which is supported only under UNI 4.0. The alerting
     message contains no mandatory IEs; only the  message  header
     is filled in.

     qcc_create_call_proceeding() creates  the  structure  for  a
     call  proceeding  message,  which contains no mandatory IEs.
     Only the message header is filled in.

     qcc_create_connect() creates the  structure  for  a  connect
     message,  which  also contains no mandatory IEs. Again, only
     the required header is filled in.   The  same  is  true  for
     qcc_create_connect_ack.

     qcc_create_release() creates  a  release  message  structure
     containing  a  cause  IE,  for which the user must pass in a
     cause value.  The  possible  values  can  be  found  in  the
     <atm/qccdefs.h>  header  file.  By default, no diagnostic is
     included and the user location is assigned.

     qcc_create_release_complete() creates the  structure  for  a
     release  complete  message, which contains no mandatory IEs.
     Only the message header is filled in.

     qcc_create_status_enquiry() creates a status enquiry message
     structure, which contains no mandatory IEs. Only the message
     header is filled in.

     qcc_create_status() builds a status message structure,  con-
     taining  two  mandatory  IEs: call state and cause. The user
     should pass in value for both the callstate and  the  cause;
     possible  values  may be found in the <atm/qccdefs.h> header
     file. In the cause IE, no diagnostic  is  included  and  the
     user location is assigned.

     qcc_create_notify() builds a notify message structure, which
     is only supported under UNI 4.0. The message contains a sin-
     gle mandatory IE, the notification indicator, which contains
     a  buffer  of  user-specified  data. The maximum size of the
     buffer is 16 bytes, defined as  QCC_MAX_NOTIFICATION_LEN  in
     <atm/qcc.h>.  The  user should allocate a buffer and pass in
     the buffer length, contentlen, and a pointer to the  buffer,
     contentp.

     qcc_create_restart() creates a  restart  message  structure,
     containing  the  mandatory restart indicator IE, and option-
     ally the connection identifier IE.  The user should pass  in
```

CODE EXAMPLE 5-2 `qcc_create(9F)` Man Page *(Continued)*

```
     a     value    for    the    restart    indicator,    either
     RESTART_INDICATED_VC or RESTART_ALL_VCS. If a  non-zero  vci
     parameter is passed in, the connection identifier IE is also
     included in the message, using a default vpci of 0  and  the
     vci parameter value.

     qcc_create_add_party() constructs  an  add  party   message
     structure.   It  includes  the mandatory called party number
     and endpoint reference IEs.   The  user  should  pass  in  a
     pointer  to  the  called  number  and  an endpoint reference
     value; for the called party  number,  ATM  Endsystem  (NSAP)
     address type is assumed.

     qcc_create_add_party_ack() fills in an add party ack message
     structure  with  the  endpoint  reference  IE. The endpt_ref
     parameter value is used.

     qcc_create_party_alerting() creates a party alerting message
     structure  with  the  endpoint  reference IE, which uses the
     endpt_ref parameter.  This message type  is  only  supported
     under UNI 4.0.

     qcc_create_add_party_reject() fills the cause  and  endpoint
     reference  IEs  into an add party reject structure. The user
     should provide the cause and endpoint reference value;  pos-
     sible cause values are defined in the <atm/qccdefs.h> header
     file. By default, no diagnostic is  included  and  the  user
     location is assigned in the cause IE.

     qcc_create_drop_party() fills the cause and endpoint  refer-
     ence  IEs  into a drop party structure. The user should pass
     in the cause and endpoint reference values;  possible  cause
     values  are  defined  in the <atm/qccdefs.h> header file. By
     default, no diagnostic is included and the user location  is
     assigned in the cause IE.

     qcc_create_drop_party_ack() fills in only the mandatory end-
     point reference IE, requiring the endpt_ref parameter.

     qcc_create_leaf_setup_fail() creates a leaf setup fail  mes-
     sage  structure,  with  three  mandatory  IEs.  The cause IE
     requires the cause parameter, which should  be  one  of  the
     cause  values  defined in <atm/qccdefs.h>; the called number
     IE requires the destination ATM address, dst_addrp; and  the
     leaf  number  IE requires the leaf_num parameter.  This mes-
     sage is only supported under UNI 4.0.
```

CODE EXAMPLE 5-2 qcc_create(9F) Man Page *(Continued)*

```
     qcc_create_leaf_setup_req() creates  a  leaf  setup  request
     message structure, with four mandatory IEs. Both the calling
     party and called party number IEs are  required,  using  the
     source  and  destination  ATM  addresses,  passed  in in the
     src_addrp and dst_addrp parameters, respectively.  The  leaf
     initiated  join  call  identifier IE requires the lij_callid
     parameter. The final required IE, the  leaf  number  IE,  is
     inserted  as  a  placeholder; the actual leaf number will be
     assigned and filled in by  the  q93b  driver.   It  will  be
     returned  in  the  callref  field  of  the  qcc  header of a
     leaf_setup_ack  message,  much  as  the  call  reference  is
     returned in a setup_ack message in the setup case.  Refer to
     the description of the qcc_bld_leaf_setup_req() function for
     more  details  on  this  process.  This message is only sup-
     ported under UNI 4.0.

RETURN VALUES
     All functions return 0 on success and -1 on error.

EXAMPLES
     The following code fragment creates a setup message, adds an
     optional  AAL  Parameters IE, packs the message into m_blks,
     and sends it downstream.

          #include <sys/stream.h>
          #include <atm/limits.h>
          #include <atm/qcc.h>
          #include <atm/qcctypes.h>

          char    _depends_on[] = "drv/qcc";

          void
          send_setup(queue_t *q);
          {
               mblk_t  *mp;
               char    ifname[QCC_MAX_IFNAME_LEN] = "ba0";
               int     calltag = 0x1234;
               int     forward_sdusize = 0x2378;
               int     backward_sdusize = 0x2378;
               qcc_msg_t          msgstruct;
               qcc_setup_t        setup;
               qcc_ie_t           iestruct;
               qcc_aal_params_t   aal;

               atm_addr_t     dst_addr = {
```

CODE EXAMPLE 5-2 qcc_create(9F) Man Page *(Continued)*

```
                    0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
                    0x08, 0x00, 0x20, 0x1a, 0xb6, 0xb9, 0x00
               };

               if ((qcc_create_setup(&setup, ifname,
                                     calltag, dst_addr)) < 0) {
                    printf("qcc_create_setup failed\n");
                    exit (-1);
               }

               msgstruct.type = QCC_SETUP;
               msgstruct.msg.setup = &setup;

               aal.type = AAL_TYPE_5;
               aal.info.aal5.forward_max = forward_sdusize;
               aal.info.aal5.backward_max = backward_sdusize;
               aal.info.aal5.mode = MESSAGE_MODE;
               aal.info.aal5.sscs_type = SSCS_TYPE_NULL;

               iestruct.type = QCC_AAL_PARAMETERS;
               iestruct.ie.aal_params = &aal;

               if ((qcc_set_ie(&msgstruct, &iestruct)) < 0) {
                    printf("qcc_set_ie failed\n");
                    exit (-1);
               }

               if ((mp = qcc_pack_setup(&setup)) == NULL) {
                    printf("qcc_pack_setup failed\n");
                    exit (-1);
               }

               if (putq(q, mp) < 0) {
                    perror("putq");
                    exit (-1);
               }
          }

SEE ALSO
     qcc_util(3), qcc_set_ie(9F),  qcc_pack(9F),  qcc_unpack(9F),
     qcc_parse(9F), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.
```

CODE EXAMPLE 5-2 `qcc_create(9F)` Man Page *(Continued)*

```
NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.  These message types, if sent on an interface
     configured for UNI 3.0 or 3.1, will be discarded by the q93b
     driver  and  will  not  be sent out to the network.  The UNI
     4.0-specific messages are Alerting, Notify, Party  Alerting,
     Leaf  Setup Fail, and Leaf Setup Request, and are identified
     in the applicable function descriptions.
```

qcc_pack(9F)

CODE EXAMPLE 5-3 qcc_pack(9F) Man Page

```
qcc_pack(9F)      Kernel Functions for Drivers       qcc_pack(9F)



NAME
     qcc_pack,         qcc_pack_setup,         qcc_pack_alerting,
     qcc_pack_call_proceeding,                  qcc_pack_connect,
     qcc_pack_connect_ack,                      qcc_pack_release,
     qcc_pack_release_complete,                  qcc_pack_status,
     qcc_pack_status_enq,   qcc_pack_notify,    qcc_pack_restart,
     qcc_pack_restart_ack,                    qcc_pack_add_party,
     qcc_pack_add_party_ack,             qcc_pack_party_alerting,
     qcc_pack_add_party_reject,              qcc_pack_drop_party,
     qcc_pack_drop_party_ack,           qcc_pack_leaf_setup_fail,
     qcc_pack_leaf_setup_req  -  encode  Q.2931 message structure
     information and pack into streams buffers

SYNOPSIS
     cc -DKERNEL -D_KERNEL [ flag ... ] file ...

     #include <atm/types.h>
     #include <atm/qcc.h>

     char _depends_on[] = "drv/qcc";

     mblk_t *qcc_pack_setup(qcc_setup_t *msgp);

     mblk_t *qcc_pack_alerting(qcc_alerting_t *msgp);

     mblk_t *qcc_pack_call_proceeding(qcc_call_proc_t *msgp);

     mblk_t *qcc_pack_connect(qcc_connect_t *msgp);

     mblk_t *qcc_pack_connect_ack(qcc_connect_ack_t *msgp);

     mblk_t *qcc_pack_release(qcc_release_t *msgp);

     mblk_t *qcc_pack_release_complete(
               qcc_release_complete_t *msgp);
```

CODE EXAMPLE 5-3 `qcc_pack(9F)` Man Page *(Continued)*

```
     mblk_t *qcc_pack_status_enq(qcc_status_enq_t *msgp);

     mblk_t *qcc_pack_status(qcc_status_t *msgp);

     mblk_t *qcc_pack_notify(qcc_notify_t *msgp);

     mblk_t *qcc_pack_restart(qcc_restart_t *msgp);

     mblk_t *qcc_pack_restart_ack(qcc_restart_ack_t *msgp);

     mblk_t *qcc_pack_add_party(qcc_add_party_t *msgp);

     mblk_t *qcc_pack_add_party_ack(qcc_add_party_ack_t *msgp);

     mblk_t *qcc_pack_party_alerting(qcc_party_alerting_t *msgp);

     mblk_t *qcc_pack_add_party_reject(
               qcc_add_party_reject_t *msgp);

     mblk_t *qcc_pack_drop_party(qcc_drop_party_t *msgp);

     mblk_t *qcc_pack_drop_party_ack(qcc_drop_party_ack_t *msgp);

     mblk_t *qcc_pack_leaf_setup_fail(
               qcc_leaf_setup_fail_t *msgp);

     mblk_t *qcc_pack_leaf_setup_req(qcc_leaf_setup_req_t *msgp);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The -DKERNEL and -D_KERNEL flags must be included  to  indi-
     cate  that  the  application should run in kernel space, and
     the qcc driver must be loaded (this requirement is expressed
     in  the  code  using  the  "depends_on"  line  shown  in the
     synopsis).

DESCRIPTION
     These functions take message structures as input and  encode
     the  information  contained  in  the  structure  to create a
     Q.2931 message, which is then packed into mblk_t structures.
     The  Q.2931  protocol  is  used  for  ATM signalling; a full
     description of the message format and use can  be  found  in
```

CODE EXAMPLE 5-3 qcc_pack(9F) Man Page *(Continued)*

```
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0. The encoded messages will conform to the ver-
     sion  of  the  UNI  Specification which is configured on the
     indicated interface. The functions may be used by  processes
     which are running in kernel space.

     Message structures should be filled using the qcc_create(9F)
     and  qcc_set_ie(9F)  functions before calling qcc_pack func-
     tions.

     In general, no error checking is performed on the data  that
     is  passed  in.   Whatever  data is contained in the message
     structure will be placed  in  the  encoded  message  without
     examination.

     Each function requires 1 parameter: msgp, which is a pointer
     to the appropriate message structure.

     Two mblk_t structures are allocated and linked  by  each  of
     the  functions  (their  format  is  shown  in  the following
     diagram).  The  pointer  that  is  returned  points  to  the
     M_PROTO  block,  and  may then be passed downstream with the
     putq(9F) command.

                      M_PROTO                           M_DATA
                ______________      ___________________________
           --->|              |--->|     |               |     |
               |  IF_Name     |    |  Q.2931 Message     |     |
               |  Call_ID     |    |     |               |     |
               |  Type        |    |     |               |     |
               |  Error       |    |     |  Information  |     |
               |  Call_Tag    |    | (9) |  Elements     | (16)|
               |______________|    |_____|_______________|_____|

     The information in the message structure passed in  to  each
     function  is  used to fill in the data portions of these two
     mblks.

RETURN VALUES
     All functions return a pointer to an mblk_t. If the function
     is not successful, the pointer will be NULL.

EXAMPLES
     For an example using qcc_pack_setup, see the example in  the
     qcc_create(9F) man page.
```

CODE EXAMPLE 5-3 qcc_pack(9F) Man Page *(Continued)*

```
SEE ALSO
     qcc_util(3), qcc_create(9F), qcc_set_ie(9F), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message types will be ignored by the
     q93b driver if used on an interface which is configured  for
     UNI 3.0 or 3.1.  The UNI 4.0-specific messages are Alerting,
     Notify, Party Alerting, Leaf  Setup  Fail,  and  Leaf  Setup
     Request.
```

qcc_parse(9F)

CODE EXAMPLE 5-4 qcc_parse(9F) Man Page

```
qcc_parse(9F)     Kernel Functions for Drivers      qcc_parse(9F)



NAME
     qcc_parse,       qcc_parse_setup,        qcc_parse_alerting,
     qcc_parse_call_proceeding,                qcc_parse_connect,
     qcc_parse_release,             qcc_parse_release_complete,
     qcc_parse_status_enquiry,                  qcc_parse_notify,
     qcc_parse_status, qcc_parse_restart,  qcc_parse_restart_ack,
     qcc_parse_add_party,                qcc_parse_add_party_ack,
     qcc_parse_party_alerting,        qcc_parse_add_party_reject,
     qcc_parse_drop_party,              qcc_parse_drop_party_ack,
     qcc_parse_leaf_setup_fail, qcc_parse_leaf_setup_req -  parse
     Q.2931 messages

SYNOPSIS
     cc -DKERNEL -D_KERNEL [ flag ... ] file ...

     #include <atm/types.h>
     #include <atm/qcc.h>

     char _depends_on[] = "drv/qcc";

     int qcc_parse_setup(mblk_t *mp, int *vcip,
           int *forward_sdusizep, int *backward_sdusizep,
           atm_addr_t *src_addrp, atm_addr_t *dst_addrp,
           int *sapp, int *endpt_refp);

     int qcc_parse_alerting(mblk_t *mp, int *vcip,
           int *endpt_refp);

     int qcc_parse_call_proceeding(mblk_t *mp, int *vcip,
           int *endpt_refp);

     int qcc_parse_connect(mblk_t *mp, int *vcip,
           int *forward_sdusizep, int *backward_sdusizep,
           int *endpt_refp);

     int qcc_parse_release(mblk_t *mp, int *causep);
```

CODE EXAMPLE 5-4 `qcc_parse(9F)` Man Page *(Continued)*

```
     int qcc_parse_release_complete(mblk_t *mp,
           int *causep);

     int qcc_parse_status_enquiry(mblk_t *mp,
           int *endpt_refp);

     int qcc_parse_status(mblk_t *mp, int *callstatep,
           int *causep, int *endpt_refp, int *endpt_statep);

     int qcc_parse_notify(mblk_t *mp, int *contentlenp,
           u_char *contentp, int *endpt_refp);

     int qcc_parse_restart(mblk_t *mp, int *vcip,
           int *rstallp);

     int qcc_parse_restart_ack(mblk_t *mp, int *vcip,
           int *rstallp);

     int qcc_parse_add_party(mblk_t *mp, int *forward_sdusizep,
           int *backward_sdusizep, atm_address_t *src_addrp,
           atm_address_t *dst_addrp, int *sapp, int *endpt_refp);

     int qcc_parse_add_party_ack(mblk_t *mp, int *endpt_refp);

     int qcc_parse_party_alerting(mblk_t *mp, int *endpt_refp);

     int qcc_parse_add_party_reject(mblk_t *mp, int *causep,
           int *endpt_refp);

     int qcc_parse_drop_party(mblk_t *mp, int *causep,
           int *endpt_refp);

     int qcc_parse_drop_party_ack(mblk_t *mp, int *causep,
           int *endpt_refp);

     int qcc_parse_leaf_setup_fail(mblk_t *mp, int *causep,
           atm_address_t *dst_addrp, int *leaf_nump);

     int qcc_parse_leaf_setup_req(mblk_t *mp,
           atm_address_t *src_addrp, atm_address_t *dst_addrp,
           int *lij_callidp, int *leaf_nump);

MT-LEVEL
     Safe.
```

CODE EXAMPLE 5-4 qcc_parse(9F) Man Page *(Continued)*

```
AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The -DKERNEL and -D_KERNEL flags must be included  to  indi-
     cate  that  the  application should run in kernel space, and
     the qcc driver must be loaded (this requirement is expressed
     in  the  code  using  the  "depends_on"  line  shown  in the
     synopsis).

DESCRIPTION
     These functions parse the various messages that make up  the
     Q.2931  protocol  which  is used for ATM signalling.  A full
     description of the message format and use can  be  found  in
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0. Messages conforming to both versions will  be
     parsed.   The  functions  may be used by processes which are
     running in kernel space.

     Each function requires a minimum of 1 parameter:  mp,  which
     is  a  pointer  to a mblk_t structure, and is extracted from
     the following structure:

                      M_PROTO                           M_DATA
                ____________      ____________________________
           --->|             |--->|      |               |     |
               |  IF_Name    |    |  Q.2931 Message      |     |
               |  Call_ID    |    |      |               |     |
               |  Type       |    |      |               |     |
               |  Error      |    |      | Information   |     |
               |  Call_Tag   |    | (9)  | Elements      | (16)|
               |_____________|    |______|_______________|_____|

     When a message is received from the q93b  driver  using  the
     getq(9F)  function,  a  pointer  to  the M_PROTO block shown
     above is returned.  However,  the  q93b  message  which  is
     parsed is contained in the M_DATA block, so the first param-
     eter passed to a  qcc_parse  function  must  be  mp->b_cont,
     where  mp  is  the  pointer received by getq().  The M_PROTO
     block data may be examined to determine  the  message  type,
     which indicates the parsing function that should be called.

     Other parameters for each function depend  on  the  type  of
     information  that is available in each message type.  In all
     cases, certain IEs are examined in each  message,  as  indi-
     cated  below.  If those IEs exist, the data that is expected
     from them is retrieved, but no error message is sent if they
```

CODE EXAMPLE 5-4 qcc_parse(9F) Man Page *(Continued)*

```
do  not  exist;  the value of the parameter is set to -1 for
any data that was expected from that particular  IE.   Also,
IEs that are not expected are ignored. If the user wishes to
ignore any of the parameters of a parse function, passing in
a  NULL  pointer for that parameter is allowed so that space
need not be allocated for the unnecessary parameter.

qcc_parse_setup() parses a setup message containing the fol-
lowing  Information  Elements: AAL parameters, ATM user cell
rate, broadband bearer capability, called party number, cal-
ling  party number, quality of service parameter, connection
identifier, broadband higher layer information, and endpoint
reference.  The  endpoint  reference  IE is only included in
setup messages for point-to-multipoint calls.  The following
table  matches  the  data that is retrieved from the message
with the IE from which it is parsed.

       DATA RETRIEVED              INFORMATION ELEMENT
       vci                         connection identifier
       forward sdusize             AAL parameters
       backward sdusize            AAL parameters
       source address              calling party number
       destination address         called party number
       sap                         broadband higher layer
       endpoint reference id       endpoint reference

qcc_parse_alerting() parses an alerting message. The  alert-
ing  message  is new in UNI 4.0; if received on an interface
configured for uni 3.0 or 3.1, it will  be  dropped  by  the
q93b driver.  The IEs examined by this function are the con-
nection identifier IE, from which the vci is parsed, and the
endpoint reference IE, from which the endpt_ref parameter is
parsed.  The endpoint  reference  IE  is  only  included  in
alerting messages for point-to-multipoint calls.

qcc_parse_call_proceeding() parses a call proceeding message
containing  a connection identifier IE, which is used to set
the value of vci, and an endpoint reference IE, setting  the
value  of  endpt_ref.  The  endpoint  reference  IE  is only
included in call proceeding messages for point-to-multipoint
calls.

qcc_parse_connect() parses a connect message  containing  an
AAL  parameters IE, setting the forward and backward sdusize
values, a connection identifier IE,  setting  the  value  of
vci,  and  an  endpoint  reference  IE, setting the value of
```

CODE EXAMPLE 5-4 qcc_parse(9F) Man Page *(Continued)*

```
     endpt_ref. The endpoint reference IE  is  only  included  in
     connect messages for point-to-multipoint calls.

     qcc_parse_release() parses a cause  IE,  setting  the  cause
     value.  A listing of the possible values can be found in the
     <atm/qcc.h>   header   file.   The   same   is   true    for
     qcc_parse_release_complete.

     qcc_parse_status_enquiry() parses a status  enquiry  message
     containing  an  endpoint  reference IE, setting the value of
     endpt_ref. The endpoint reference IE is only  included  when
     enquiring about a party state in a point-to-multipoint call.

     qcc_parse_status() parses a status message.   The  IEs  that
     are  parsed  are  call state, cause, endpoint reference, and
     endpoint state. The call state and cause IEs are used to set
     the  values  of the parameters callstate and cause; possible
     values for both parameters may be found in  the  <atm/qcc.h>
     header  file.  The endpoint reference and endpoint state IEs
     will be  used  to  set  the  values  of  the  endpt_ref  and
     endpt_state  parameters;  they are included if an enquiry is
     made about a party state in a point-to-multipoint call or to
     report an error condition in a point-to-multipoint call.

     qcc_parse_notify() parses a notify message,  which  is  only
     supported under UNI 4.0. The notification indicator and end-
     point reference IEs are parsed; from the notification  indi-
     cator,  the  contentlenp  and contentp parameters are filled
     in, with the maximum buffer size copied being 16 bytes.   If
     the  size  contained in the message is greater than 16 bytes
     (QCC_MAX_NOTIFICATION_LEN,  defined  in  <atm/qcc.h>),   the
     first 16 bytes are copied, contentlenp is set to contain the
     copied length of 16 bytes, and the  overflow  flag  is  set.
     From  the  endpoint  reference  IE, endpt_refp is filled in.
     The endpoint reference  IE  is  only  present  on  point-to-
     multipoint calls.

     qcc_parse_restart() parses a restart message containing  two
     possible  IEs:  connection identifier and restart indicator.
     The restart indicator IE is used to set the value of rstall;
     this  parameter  indicates  whether  a particular vci or all
     vcis are to be restarted  (rstall  =  1  implies  all  vcis,
     rstall  = 0 implies a particular vci).  The connection iden-
     tifier identifies the particular vci.   In  this  case,  the
     value  of  the parameter vci is set to 0 if there is no con-
     nection identifier IE  in  the  message.   The  same  format
```

CODE EXAMPLE 5-4 qcc_parse(9F) Man Page *(Continued)*

```
     applies to the qcc_parse_restart_ack() function.

     qcc_parse_add_party() parses an add party message containing
     sever  possible  IEs.  They  include AAL parameters, calling
     party number, called party number,  broadband  higher  layer
     information,  and  endpoint  reference.  The following table
     matches the data that is retrieved from the message with the
     IE from which it is parsed.

            DATA RETRIEVED              INFORMATION ELEMENT
            forward sdusize             AAL parameters
            backward sdusize            AAL parameters
            source address              calling party number
            destination address         called party number
            sap                         broadband higher layer
            endpoint reference id       endpoint reference

     qcc_parse_add_party_ack()  extracts  an  endpoint  reference
     value  from  the  endpoint  reference IE in an add party ack
     message.

     qcc_parse_party_alerting() extracts  an  endpoint  reference
     value  from  the  endpoint  reference IE in a party alerting
     message.  This message is specific to UNI 4.0.

     qcc_parse_add_party_reject() parses an add party reject mes-
     sage  possibly containing a cause IE, from which it extracts
     the cause value, and an endpoint reference IE, from which it
     extracts the endpoint reference value. Possible cause values
     may be found in the header file <atm/qcc.h>.

     qcc_parse_drop_party() extracts an endpoint reference  value
     and  a cause value from those respective IEs in a drop party
     message. Possible cause values may be found  in  the  header
     file    <atm/qcc.h>.    The    same    parsing    applies    to
     qcc_parse_drop_party_ack().

     qcc_parse_leaf_setup_fail() extracts a cause value  (defined
     in  <atm/qcc.h>)  from  the  cause IE; a destination address
     from the called number IE; and a leaf number from  the  leaf
     number  IE.  The  leaf setup fail message is specific to UNI
     4.0.

     qcc_parse_leaf_setup_req() parses a leaf setup request  mes-
     sage,  which is specific to UNI 4.0.  The calling number and
     called number IEs are parsed, yielding the source and desti-
```

CODE EXAMPLE 5-4 qcc_parse(9F) Man Page *(Continued)*

```
     nation  ATM  addresses,  respectively; in addition, the leaf
     initiated join call identifier IE is parsed  to  obtain  the
     leaf initiated join callid, and the leaf number IE is parsed
     for the leaf number.

RETURN VALUES
     All functions return 0 on success and -1 on error.

EXAMPLES
     The following code fragment receives and parses a setup mes-
     sage.

          #include <sys/stream.h>
          #include <atm/qcc.h>
          #include <atm/limits.h>

          char    _depends_on[] = "drv/qcc";

          void
          wait_for_setup(queue_t *q);
          {
               int            vci;
               int            forward_sdusize;
               int            backward_sdusize;
               int            sap;
               atm_addr_t     src_addr;
               atm_addr_t     dst_addr;
               mblk_t         *mp;
               qcc_hdr_t      *hdrp;

               do {
                    if !(mp = getq(q)) {
                         perror("getq");
                         exit (-1);
                    }
                    hdrp = (qcc_hdr_t *)mp;
               } while (hdrp->type != QCC_SETUP);

               qcc_parse_setup(mp->b_cont, &vci, &forward_sdusize,
                                &backward_sdusize, &src_addr,
                                &dst_addr, &sap, NULL);
               printf("parse_setup: vci = 0x%x, sap = 0x%x0, vci, sap);
          }

SEE ALSO
     qcc_util(3), qcc_bld(9F), q93b(7)
```

CODE EXAMPLE 5-4 `qcc_parse(9F)` Man Page *(Continued)*

```
     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message  types,  if  received  on an
     interface configured for UNI 3.0 or 3.1, will  be  discarded
     by  the  q93b  driver  and  will  not be sent up to the user
     applications.  The UNI 4.0-specific messages  are  Alerting,
     Notify, Party Alerting, Leaf Setup Fail, and Leaf Setup Req,
     and are identified in the applicable function descriptions.
```

qcc_set_ie(9F)

CODE EXAMPLE 5-5 qcc_set_ie(9F) Man Page

```
qcc_set_ie(9F)    Kernel Functions for Drivers     qcc_set_ie(9F)



NAME
     qcc_set_ie - add or update Information Elements in a  Q.2931
     message structure

SYNOPSIS
     cc -DKERNEL -D_KERNEL [ flag ... ] file ...

     #include <atm/qcc.h>
     #include <atm/qcctypes.h>

     char _depends_on[] = "drv/qcc";

     int qcc_set_ie(qcc_msg_t *msgp, qcc_ie_t *iep);

MT-LEVEL
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.
     The -DKERNEL and -D_KERNEL flags must be included  to  indi-
     cate  that  the  application should run in kernel space, and
     the qcc driver must be loaded (this requirement is expressed
     in  the  code  using  the  "depends_on"  line  shown  in the
     synopsis).

DESCRIPTION
     This function adds a new or changes an existing  Information
     Element in Q.2931 messages.  The Q.2931 protocol is used for
     ATM signalling.  A full description of  the  message  format
     and  use can be found in the ATM Forum's User Network Inter-
     face Specification, V3.0 or V3.1. The function may  be  used
     by processes which are running in kernel space.

     A message  structure  should  first  be  created  using  the
     appropriate  qcc_create(9F)  function call.  IEs may then be
```

CODE EXAMPLE 5-5 qcc_set_ie(9F) Man Page *(Continued)*

```
     added or changed using qcc_set_ie.  When the message  struc-
     ture  has  been  completely  specified,  the  corresponding
     qcc_pack(9F) function should be called to translate the mes-
     sage structure into the correct encoded format, contained in
     mblk_t structures which may be passed to the putq(9F)  func-
     tion.

     In general, no error checking is performed on the data  that
     is  passed in.  Whatever data is passed in will be placed in
     the message that is built  without  examination.   The  user
     should  insure that the values passed in in the IE structure
     conform with the UNI version (3.0 or 3.1) that is running.

     The function requires 2 parameters: msgp, which is a pointer
     to  the  appropriate  message structure; and iep, which is a
     pointer to  the  new  IE  structure.  The  message  and  IE
     structure  types  are defined in the <atm/qcctypes.h> header
     file.

     The structure to which msgp points must be allocated by  the
     calling  user.   The structure pointed to by iep should have
     the desired values filled in to its fields, and the  "valid"
     field should be set to 1.  A value of 0 in the "valid" field
     indicates that the IE should not be included in the message.

     The fields of each Information Element structure  and  their
     interpretations  are  described in the following paragraphs.
     Possible  values  for  IE  fields   are   defined   in   the
     <atm/qccdefs.h> header file.

     qcc_aal_params_t
          Currently, the only ATM Adaptation Layer  supported  on
          SunATM products is AAL 5.  However, to allow for future
          changes, the aal parameters ie type consists of a field
          identifying  the aal and a union of structures for each
          aal, called "info."  The aal  5  structure  contains  4
          fields: forward_max and backward_max for the SDU sizes,
          mode, and sscs_type. The sscs_type is only valid in UNI
          3.0;  therefore,  a  value of 0 for sscs_type indicates
          that that field should not be included.

     qcc_traffic_desc_t
          The ATM Traffic Descriptor IE (called User Cell Rate in
          UNI  3.0)  contains  a  large  set of traffic parameter
          values.  Two parameters  do  not  have  numeric  values
          associated;  they  are either included or not.  The are
```

CODE EXAMPLE 5-5 qcc_set_ie(9F) Man Page *(Continued)*

```
          represented by two  fields,  best_effort  and  tagging,
          that  are  either  set  to  1 if the parameter is to be
          included or set to 0  if  it  is  not.   The  remaining
          parameters  all  have  numeric  values  associated with
          them.  Since 0 is a valid value for  these  parameters,
          an  additional  field,  params,  is  included in the IE
          structure which indicates  which  of  these  should  be
          included   in   the  message.  Each  parameter  has  a
          corresponding bit in the params field, which, when set,
          indicates  that the parameter should be included. Flags
          are defined  for  this  field  in  the  <atm/qccdefs.h>
          header file.

     qcc_bbc_t
          The Broadband Bearer Capability  IE  fields  correspond
          directly to the options for this IE.  The fields are:

              class           Bearer Class
              type            Traffic Type
              timing          Timing Requirements
              clipping        Susceptibility to Clipping
              userplane       User plane connection configuration

     qcc_bhli_t
          The Broadband High Layer Information IE structure  con-
          tains 3 fields which specify the IE contents.  They are
          type, which identifies the High Layer Information Type;
          infolen,  which  indicates the number of octets of high
          layer information is to be included in the message (the
          maximum  is  8  octets),  and finally an array of bytes
          called info  which  contains  the  information  octets,
          called  info.  The octets should be placed in the first
          infolen elements of the array.

     qcc_blli_t
          The Broadband  Low  Layer  Information  IE  contains  2
          fields  to  specify the IE contents.  The first, layer,
          is an integer which specifies which layer  protocol  is
          being  specified,  layer  1,  2, or 3.  The second is a
          union, with unique structures for layer 2 and layer  3.
          For  both  layer  2 and layer 3 IEs, the protocol value
          will be examined and the correct coding format will  be
          used for that protocol.  Therefore, only the applicable
          fields from the layer structure will be  used  for  the
          specified protocol type.
```

CODE EXAMPLE 5-5 `qcc_set_ie(9F)` Man Page *(Continued)*

```
     Layer 2 fields:
         protocol     User information layer 2 protocol
         mode         Mode of operation
         windowsize   Window size (k)
         userspec     User specified layer 2 protocol
                      information


     Layer 3 fields:
         protocol     User information layer 3 protocol
         mode         Mode of operation
         pktsize      Default packet size
         windowsize   Packet window size
         userspec     User specified layer 3 protocol
                      information
         ipi          8-bit Initial Protocol Identifier for
                      ISO/IEC TR 9577
         oui          24-bit organization unique identifier
                      for ISO/IEC TR 9577 and IEEE 802.1 SNAP
         pid          16-bit protocol identifier for ISO/IEC
                      TR 9577 and IEEE 802.1 SNAP


qcc_call_state_t
     There is only one informational field in the Call State
     IE structure:  state, specifying the call state.

qcc_called_num_t
     The Called Party Number IE structure contains a  planid
     field,  which  specifies  the Addressing/Numbering Plan
     Identification.  The Type of Number is  based  on  this
     value  as  well.   There  is  also an address field, to
     specify a 20-byte address.

qcc_called_subaddr_t
     The Called Party Subaddress  IE  structure  contains  a
     type field, which specifies the Type of Subaddress, and
     a 20-byte address field.

qcc_calling_num_t
     In addition to the 20-byte address field,  the  Calling
     Party  Number  IE  structure contains several fields to
     describe the intended interpretation  of  the  address.
     They are:

         planid         Addressing/Numbering Plan
```

CODE EXAMPLE 5-5 qcc_set_ie(9F) Man Page *(Continued)*

```
                        Identification
               presentation   Presentation indicator
               screening      Screening indicator

     qcc_calling_subaddr_t
          The structure for the Calling Party  Subaddress  IE  is
          identical to that of the Called Party Subaddress IE.

     qcc_cause_t
          The Cause IE structure contains a location field and  a
          cause  field.   In addition, it contains an array of 28
          octets, diag, for diagnostic information. The number of
          diagnostic  octets  included  in  the  array  should be
          specified in the diaglen field.

     qcc_conn_id_t
          The Connection Identifier IE structure contains a  vpci
          and  a  vci  field.  Note  that  currently,  the SunATM
          software only supports vpci 0, although any  value  may
          be  placed  in  the vpci field and will be encoded into
          the message.

     qcc_qos_t
          The Quality of Service IE has 3  informational  fields:
          codingstd,  specifying  the  Coding Standard value; and
          forward_class and backward_class, specifying  the  For-
          ward and Backward QoS Class.

     qcc_restart_ind_t
          There is only one informational field  in  the  Restart
          Indicator  IE  structure:  class,  whcih  specifies the
          class of the facility to be restarted.

     qcc_transit_t
          The Transit Network Selection IE structure contains  an
          array of up to four octets to specify the Carrier Iden-
          tification Code value.

     qcc_endpt_ref_t
          The  Endpoint  Reference  IE  structure   contains   an
          endptref  field, which specifies the endpoint reference
          value.

     qcc_endpt_state_t
          The Endpoint State IE structure contains a state field,
          which identifies the endpoint state value.
```

CODE EXAMPLE 5-5 `qcc_set_ie(9F)` Man Page *(Continued)*

```
RETURN VALUES
     The function returns 0 on success and -1 on error.

EXAMPLES
     See the Example section of the qcc_create(9F) man  page  for
     an example using qcc_set_ie.

SEE ALSO
     qcc_util(3), qcc_create(9F),  qcc_pack(9F),  qcc_unpack(9F),
     qcc_parse(9F), q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.
```

qcc_unpack(9F)

CODE EXAMPLE 5-6 qcc_unpack(9F) Man Page

```
qcc_unpack(9F)    Kernel Functions for Drivers     qcc_unpack(9F)



NAME
     qcc_unpack,      qcc_unpack_setup,      qcc_unpack_alerting,
     qcc_unpack_call_proceeding,              qcc_unpack_connect,
     qcc_unpack_connect_ack,                  qcc_unpack_release,
     qcc_unpack_release_complete,              qcc_unpack_status,
     qcc_unpack_status_enq,                    qcc_unpack_notify,
     qcc_unpack_restart,                  qcc_unpack_restart_ack,
     qcc_unpack_add_party,              qcc_unpack_add_party_ack,
     qcc_unpack_party_alerting,      qcc_unpack_add_party_reject,
     qcc_unpack_drop_party,            qcc_unpack_drop_party_ack,
     qcc_unpack_leaf_setup_fail,    qcc_unpack_leaf_setup_req   -
     decode Q.2931 messages and unpack into message structures

SYNOPSIS
     cc -DKERNEL -D_KERNEL [ flag ... ] file ...

     #include <atm/types.h>
     #include <atm/qcc.h>

     char _depends_on[] = "drv/qcc";

     int qcc_unpack_setup(qcc_setup_t *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_alerting(qcc_alerting *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_call_proceeding(qcc_call_proc_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_connect(qcc_connect_t *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_connect_ack(qcc_connect_ack_t *msgp,
           mblk_t *ctlp, mblk_t *datap);
```

CODE EXAMPLE 5-6 `qcc_unpack(9F)` Man Page *(Continued)*

```
     int qcc_unpack_release(qcc_release_t *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_release_complete(qcc_release_complete_t *
           msgp, mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_status_enq(qcc_status_enq_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_status(qcc_status_t *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_notify(qcc_notify_t *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_restart(qcc_restart_t *msgp, mblk_t *ctlp,
           mblk_t *datap);

     int qcc_unpack_restart_ack(qcc_restart_ack_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_add_party(qcc_add_party_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_add_party_ack(qcc_add_party_ack_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_party_alerting(qcc_party_alerting_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_add_party_reject(qcc_add_party_reject_t *
           msgp, mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_drop_party(qcc_drop_party_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_drop_party_ack(qcc_drop_party_ack_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_leaf_setup_fail(qcc_leaf_setup_fail_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

     int qcc_unpack_leaf_setup_req(qcc_leaf_setup_req_t *msgp,
           mblk_t *ctlp, mblk_t *datap);

MT-LEVEL
```

CODE EXAMPLE 5-6 qcc_unpack(9F) Man Page *(Continued)*

```
     Safe.

AVAILABILITY
     The functionality described in this man page is available in
     the SUNWatma package included with the SunATM adapter board.

DESCRIPTION
     These functions  take  streams  buffers  containing  encoded
     Q.2931 messages as input and decode the information, placing
     the extracted values into the appropriate message structure.
     The  Q.2931  protocol  is  used  for  ATM signalling; a full
     description of the message format and use can  be  found  in
     the  ATM Forum's User Network Interface Specification, V3.0,
     V3.1, or V4.0. Messages conforming to both versions  of  the
     UNI  standard will be decoded.  The functions may be used by
     processes which are running in kernel space.

     In general, no error checking is performed on the data  that
     is  extracted from the message.  Whatever data is found will
     be placed in the message structure without examination.

     Each function  requires  3  parameters:  msgp,  which  is  a
     pointer  to  the appropriate message structure; and ctlp and
     datap, which are pointers to mblk_t structures.

     Information extracted from the message is  filled  into  the
     message structure pointed to by msgp.  The user should allo-
     cate this structure before calling the qcc_unpack function.

     The ctlp and datap mblk_t pointers should be extracted  from
     the following structure:

                      M_PROTO                           M_DATA
                ______________      ___________________________
           --->|             |--->|     |               |     |
               |  IF_Name    |    |  Q.2931 Message     |     |
               |  Call_ID    |    |     |               |     |
               |  Type       |    |     |               |     |
               |  Error      |    |     |  Information  |     |
               |  Call_Tag   |    | (9) |  Elements     | (16)|
               |_____________|    |_____|_______________|_____|

     Header information is contained in the M_PROTO mblk, and the
     q93b  message  which  is  parsed  is contained in the M_DATA
     block. When a message is received from the q93b driver using
     the  getq(9F) function, a pointer to the M_PROTO block shown
```

CODE EXAMPLE 5-6 qcc_unpack(9F) Man Page *(Continued)*

```
     above is returned. If that pointer is called mp, the pointer
     to  the  M_DATA  mblk  will be mp->b_cont. The M_PROTO block
     data may be examined to determine the  message  type,  which
     indicates the parsing function that should be called.

RETURN VALUES
     All functions return 0 on  success  and  -1  on  error.  The
     returned message structure contains an entry for each possi-
     ble Information Element for that message type; if an  Infor-
     mation Element is found in the received message, the "valid"
     field for that IE will be set to  1.   If  the  IE  was  not
     found, the "valid" field will be 0.

EXAMPLES
     The following code fragment receives  a  setup  message  and
     prints elements in the message structure.

          #include <sys/stream.h>
          #include <atm/types.h>
          #include <atm/qcc.h>
          #include <atm/limits.h>

          char    _depends_on[] = "drv/qcc";

          void
          wait_for_setup(queue_t *q);
          {
               int            vci = -1;
               int            sap = -1;
               mblk_t         *mp;
               qcc_hdr_t      *hdrp;
               qcc_setup_t    setup;

               do {
                    if ((mp = getq(q)) == NULL) {
                         perror("getq");
                         exit (-1);
                    }
                    hdrp = (qcc_hdr_t *)mp;
               } while (hdrp->type != QCC_SETUP);

               if ((qcc_unpack_setup(&setup, mp, mp->b_cont)) < 0) {
                    printf("unpack_setup failed\n");
                    exit (-1);
               }
               if (setup.conn_id.valid)
```

CODE EXAMPLE 5-6 qcc_unpack(9F) Man Page *(Continued)*

```
                    vci = setup.conn_id.vci;
               if (setup.bhli.valid)
                    memcpy((caddr_t) &sap,
                           (caddr_t) setup.bhli.info, 4);

               printf("parse_setup: vci=0x%x, sap=0x%x\n",
                       vci, sap);
          }

SEE ALSO
     qcc_util(3), qcc_create(9F),  qcc_set_ie(9F),  qcc_pack(9F),
     q93b(7)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.
     "ATM User-Network Interface Specification, V3.1," ATM Forum.
     "ATM User-Network Interface Specification, V4.0," ATM Forum.

NOTES
     This API is an interim solution  until  the  ATM  Forum  has
     standardized  an API.  At that time, Sun will implement that
     API, and support for the Q.2931 Call Control library may not
     be continued.

     The additional support of the UNI 4.0 signalling  specifica-
     tion  includes  the  addition  of  several new message types
     which are not supported in the earlier versions of  the  UNI
     specification.   These  message types will be ignored by the
     q93b driver if used on an interface which is configured  for
     UNI 3.0 or 3.1.  The UNI 4.0-specific messages are Alerting,
     Notify, Party Alerting, Leaf  Setup  Fail,  and  Leaf  Setup
     Request.
```

CHAPTER 6

Maintenance Commands

The man pages in this section describe the SunATM commands that are used chiefly for system maintenance and administration purposes.

TABLE 6-1 Maintenance Command Man Pages

Man Page	Description	Page Number
`aarsetup(1M)`	ATM Address Resolution Table setup program	page 168
`aarstat(1M)`	Display Classical IP ATM address resolver status	page 171
`atmadmin(1M)`	ATM configuration program	page 173
`atmarp(1M)`	ATM to IP address resolution	page 178
`atmgetmac(1M)`	Get the MAC address assigned to an ATM interface	page 180
`atmreg(1M)`	ATM address registration	page 181
`atmsetup(1M)`	Configure an ATM device	page 183
`atmsnmpd(1M)`	ATM SNMP agent daemon	page 185
`atmsnoop(1M)`	Capture and inspect ATM network packets	page 188
`atmspeed(1M)`	Get and set the total link bandwidth of an ATM device	page 192
`atmstat(1M)`	Display ATM network interface information	page 194
`ilmid(1M)`	ATM Address Registration daemon	page 202
`lanearp(1M)`	MAC to ATM address resolution	page 205
`lanesetup(1M)`	LAN Emulation setup program	page 208
`lanestat(1M)`	Display status of LAN Emulation over ATM	page 210
`qccstat(1M)`	Display Q.2931 call control information	page 214

aarsetup(1M)

CODE EXAMPLE 6-1 aarsetup(1M) Man Page

```
aarsetup(1M)          Maintenance Commands           aarsetup(1M)



NAME
     aarsetup - ATM Address Resolution Table setup program

SYNOPSIS
     /etc/opt/SUNWatm/bin/aarsetup [ -nkpv ] [ filename ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     The aarsetup program reads a local ATM to IP address resolu-
     tion table from the /etc/aarconfig file and loads the infor-
     mation into the kernel.  In addition, aarsetup  will  deter-
     mine whether it is executing on the client or the server and
     will configure the Classical  IP  kernel  modules  appropri-
     ately.

     If an ATM ARP server exists on a subnet,  the  configuration
     file  on clients need only contain the system's local infor-
     mation and the server information.  If an ATM ARP server  is
     not  being  used, each system's configuration file must con-
     tain IP/ATM address resolution information  for  every  host
     which it needs to contact. See the aarconfig(4) man page for
     details on the format of the configuration file.

     By default, the /etc/aarconfig file is read  and  downloaded
     into  the  local kernel table on startup.  If the configura-
     tion file is modified later, aarsetup must be rerun to  load
     the new information into the kernel.

OPTIONS
     -n        Only parse the configuration  table.   Using  this
               option,  the  syntax  and information in the table
               can be checked to verify that it is acceptable  to
               the  aarsetup  program without actually attempting
               to download any data.  Physical interface informa-
```

CODE EXAMPLE 6-1 `aarsetup(1M)` Man Page *(Continued)*

```
                tion  entered  in the table is compared with known
                configured interfaces; IP addresses must be on the
                correct  subnet  for  the  corresponding  physical
                interface in an entry.  In order to do this check-
                ing,  the  physical  interface must be configured.
                The -k option will omit the network checks.  Error
                messages  will  be  printed  if  any  problems are
                encountered.

     -k         Only parse the configuration  table,  but  do  not
                check  configured  interfaces.  Using this option,
                only the syntax of the configuration  is  checked;
                no  verification of IP address information is per-
                formed.  This enables a check of the configuration
                file before the physical interfaces have been con-
                figured.

     -p         Prints to the standard output  the  table  entries
                from  the  configuration  file,  with all variable
                expressions expanded. Does not download any infor-
                mation into the kernel.

     -v         Verbose mode.  Additional information is printed.

     filename   A filename may be specified to download  a  confi-
                guration file other than /etc/aarconfig.  Standard
                input, indicated with a hyphen  `-',  is  a  legal
                value for filename if the -n option is being used.

FILES
     /etc/aarconfig      ATM to IP  address  registration  confi-
                         guration  file.   Contains entries which
                         specify ATM and  IP  address  pairs  for
                         systems.

SEE ALSO
     aarconfig(4)

     M. Laubach, RFC 1577: Classical IP and ARP over ATM, Network
     Working Group.

NOTES
     In this context, "server" and "client" refer to an  ATM  ARP
     server  and  nodes  on  the  subnet which it serves, respec-
     tively.
```

CODE EXAMPLE 6-1 `aarsetup(1M)` Man Page *(Continued)*

```
     aarsetup SHOULD NOT be put into  the  background  (i.e.  run
     with the command 'aarsetup &'). When executed, aarsetup will
     first perform some essential first steps,  then  put  itself
     into the background without user intervention.
```

aarstat(1M)

CODE EXAMPLE 6-2 aarstat(1M) Man Page

```
aarstat(1M)           Maintenance Commands           aarstat(1M)



NAME
     aarstat - display Classical IP ATM address resolver status

SYNOPSIS
     /etc/opt/SUNWatm/bin/aarstat interface

     /etc/opt/SUNWatm/bin/aarstat -a

AVAILABILITY
     SUNWatm

DESCRIPTION
     aarstat displays information about the state of the  Classi-
     cal  IP  protocol on an ATM interface.  The information pro-
     vided may be used to debug  configuration  problems,  or  to
     verify successful bring-up of a Classical IP interface.

     The only parameter is the physical interface, which will  be
     of the form baN, where N is the instance number. Optionally,
     the -a flag may be  used  to  request  information  for  all
     interfaces.

     The following fields will be displayed for all Classical  IP
     interfaces:

     setup_state    The state of the Classical IP setup  program,
                    aarsetup.  The possible values are setup-not-
                    run, which means that aarsetup has  not  been
                    run  successfully  for this interface; setup-
                    started,  which  means   that   aarsetup   is
                    currently   running;   setup-finished,  which
                    means that  aarsetup  has  successfully  com-
                    pleted;  and  interface-defunct,  which means
                    that the interface has been partially  uncon-
                    figured by removing its entries from the con-
                    figuration files  and  re-running  aarconfig.
```

CODE EXAMPLE 6-2 `aarstat(1M)` Man Page *(Continued)*

```
                    Interfaces  whose  state is interface-defunct
                    will be removed from the  kernel  on  reboot,
                    assuming that the configuration files are not
                    changed.

     arpcsmode      The mode in which the Classical  IP  software
                    is  running.  The  possible values are stand-
                    alone,     server-being-modified,     server,
                    client-being-modified, and client. The first,
                    stand-alone, indicates  that  the  system  is
                    running  as an ATM ARP client with no ATM ARP
                    server configured.  server-being-modified and
                    client-being-modified  indicate that aarsetup
                    is currently running on the system; the  con-
                    figuration  is  not complete. Finally, server
                    and client indicate that the system is an ATM
                    ARP server or client, respectively.

     interface_state
                    The state  of  the  interface.  The  possible
                    values are up and down.

     The following additional fields will be printed  on  systems
     running as ATM ARP clients:

     server_state   The state of the connection to  the  ATM  ARP
                    server.  The  possible  values for this field
                    are no-connection, connecting, connected, and
                    closing-connection,  referring  to  phases of
                    Q.2931 call control. When an interface is  up
                    and  running  Classical  IP, the server state
                    should be connected.

     server_vci     This field will indicate the vci for the out-
                    going connection to the ATM ARP server.

     configured_server_addr
                    The atm address of the ATM ARP server.

SEE ALSO
     aarsetup(1M), aarconfig(4)
```

atmadmin(1M)

CODE EXAMPLE 6-3 atmadmin(1M) Man Page

```
atmadmin(1M)          Maintenance Commands           atmadmin(1M)



NAME
     atmadmin - ATM configuration program

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmadmin [ basedir ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     The ATM configuration program, atmadmin, is  an  interactive
     command-line  interface. The program contains a hierarchy of
     menus, which divide the configuration into six main parameter
     groups: System, Physical Layer, Signalling, ILMI, Classical IP
     and LAN Emulation. All but the System parameter group are spec-
     ific to individual SunATM interfaces, so you must configure
     the parameters in that  group  separately for each interface.
     If you prefer, you may enter and change the SunATM configuration
     information by editing  the  SunATM configuration files directly.


     By default, atmadmin looks for configuration  files  in  the
     /etc  directory.   If they are not there, the alternate path
     may be specified as basedir.  This may be desirable  if  you
     wish  to create test files, but do not want to overwrite the
     existing files in /etc.

COMMON NAVIGATION COMMANDS
     Some basic  commands  are  recognized  throughout  the  menu
     hierarchy,  and  they  may  be  used to navigate through the
     various menus. These commands are:

     m         Return to the atmadmin main menu.

     p         Return to the previous menu.
```

CODE EXAMPLE 6-3 atmadmin(1M) Man Page *(Continued)*

```
     x         Exit atmadmin.

     ?         Provide more information about the options on this
               menu.

PARAMETER GROUPS
     The atmadmin configuration  program  contains  a  series  of
     menus  where  you  can  input  or alter the configuration of
     specific SunATM software parameters. These menus, or parame-
     ter groups, are:

     System Parameter Group
               The system  parameter  group  contains  parameters
               that  are not interface-specific, but apply to the
               entire system. This group contains only  the  SNMP
               Agent Status parameter.


               Parameters  Possible Values  Default Values  Required?
               -------------------------------------------------------
               SNMP Agent  agent/not_agent  not_agent        Yes
                Status

               SNMP Agent  0 <= n <= 6535   161 or 1000      For SNMP
                UDP port                                     Agent

     Physical Layer Parameter Group
               The physical layer parameter group contains only the
               framing interface parameter.


               Parameters  Possible Values  Default Values  Required?
               -------------------------------------------------------
                Framing     SONET/SDH           SONET         Yes
               Interface

     Signalling Parameters
               The signalling parameter group contains only the UNI version
               parameter.


               Parameters  Possible Values  Default Values  Required?
               -------------------------------------------------------
               UNI Version  3.0/3.1/4.0/none  No default      Yes

     ILMI Parameters
```

CODE EXAMPLE 6-3 atmadmin(1M) Man Page *(Continued)*

```
               If your ATM switch does not support Interim Local Management
               Interface (ILMI), you can turn off the ILMI registration on
               your SunATM interface from the ILMI configuration menu.


               Parameters  Possible Values  Default Values  Required?
               -------------------------------------------------------
               Use ILMI        Yes/No           Yes           Yes

     Classical IP Parameter Group
               Several parameters define the Classical IP (CIP) configu-
               ration of a SunATM interface, and all of these parameters
               can be configured through the Classical IP parameter
               group menu.


               Parameters       Possible Values     Default Values  Required?
               --------------------------------------------------------------
               IP hostname/     Valid IP hostname   No default      For CIP
                 address          and address

               Interface        Client/Server/      No default      For CIP
                 Type            Standalone

               Local            Valid ATM address   $myaddress      For CIP
               ATM address

               ARP Server       ATM address         $localswitch_   For CIP
                                                    server          clients

               PVC             32 <= n < 1024         32           For CIP
                                                                   standalones

               Destination IP   Valid IP hostname   No default      For CIP
               hostname/address   and address                      standalones

     LAN Emulation Parameter Group
               After choosing to configure LAN Emulation (LANE) parameters,
               you will be asked to choose an existing (previously configured)
               LAN Emulation instance, or to create a new one in the LAN
               Emulation Instance menu.
```

CODE EXAMPLE 6-3 `atmadmin(1M)` Man Page *(Continued)*

```
              Parameters    Possible Values     Default Values  Required?
              ------------------------------------------------------------
              Instance      0 <= n <= 999       No default      For LANE
               Number

    Per-Instance LAN Emulation Parameters
              This menu allows you to configure the per-instance LAN
              Emulation parameters.


              Parameters    Possible Values     Default Values  Required?
              ------------------------------------------------------------
              IP hostname/  Valid IP hostname   No default      For IP
              address         and address                       over LANE

              Local         Valid ATM address   $myaddress      For LANE
              ATM address

              LECS          no_lecs/
              Indicator     lecs_present        lecs_present    For LANE

              LECS          Valid ATM address   A well-known    For LANE,
              ATM address                         address       lecs_present

              LES           Valid ATM address   No default      For LANE,
              ATM address                                       no_lecs

              Emulated      Character string    No default      For
              LAN Name                                          additional
                                                                instance
                                                                on a
                                                                physical
                                                                interface

              Additional    Yes/No              No              For LANE
              IP addresses


    Per-Additional IP address
              With this menu you can configure logical interfaces in the
              SunATM LAN Emulation environment. Logical interfaces allow
              you to assign multiple IP addresses to a single LAN Emulation
              interface. The SunATM software will associate each logical
              interface with a unique IP hostname and address. All logical
              interfaces on a given physical interface will be associated
              with the same ATM and MAC addresses.
```

CODE EXAMPLE 6-3 `atmadmin(1M)` Man Page *(Continued)*

```
               Parameters      Possible Values     Default Values  Required?
               ------------------------------------------------------------
               Minor Instance  0 <= n <= 255       None            For LANE,
                  Number                                           additional
                                                                   IP

               IP hostname/    Valid IP hostname   No default      For LANE,
               address         and address                         additional
                                                                   IP

SEE ALSO
     aarconfig(4),  aarsetup(1M),   atmconfig(4),   atmsetup(1M),
     laneconfig(4), lanesetup(1M),
```

atmarp(1M)

CODE EXAMPLE 6-4 atmarp(1M) Man Page

```
atmarp(1M)            Maintenance Commands             atmarp(1M)



NAME
     atmarp - ATM to IP address resolution

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmarp interface

     /etc/opt/SUNWatm/bin/atmarp interface [  IP  hostname  |  IP
     address ]

     /etc/opt/SUNWatm/bin/atmarp interface  -  [  ATM  address  ]
     /etc/opt/SUNWatm/bin/atmarp -a

AVAILABILITY
     SUNWatm

DESCRIPTION
     The atmarp program may be used to display ATM and IP address
     pairs  for  a  given  ATM interface.  The required parameter
     interface is a string of the form name unit, such as ba0.

     If only the interface is provided, as in the first  form  of
     the  command,  atmarp  will  print  the  ATM  address and IP
     address for that physical interface, and an entry  for  each
     resolved IP address for that interface.

     If additional information is provided, it will  be  used  to
     identify  a  device on the subnet to which interface is con-
     nected, and the corresponding address  information  will  be
     printed.   In  the  second  form, when an IP address (in the
     standard dot notation) or IP hostname is provided,  the  ATM
     address  for  that  node will be printed. In the third form,
     when an ATM address (in  the  colon-separated  octet  format
     used  in  /etc/aarconfig)  is provided, the corresponding IP
     address will be printed.  Note:  in this third form  of  the
     command,  a  hyphen (-) must be included to indicate that an
     IP hostname/address is not being provided.
```

CODE EXAMPLE 6-4 `atmarp(1M)` Man Page *(Continued)*

```
     The -a option dumps the complete ATM ARP table, listing  the
     ATM and IP address for each physical interface and a listing
     of the ATM address for each  resolved  IP  address  on  that
     interface.

EXAMPLES
     muskogee# ./atmarp ba0
     Local IP addr  = 192.168.144.108
     ATM addr = 47:00:00:00:00:00:00:00:00:00:CC:BA::08:00:20:82:BD:E1::00

     ARP Table for interface ba0:
     ----------------------------
     IP addr  = 192.168.144.108
     ATM addr = 47:00:00:00:00:00:00:00:00:00:CC:BA::08:00:20:82:BD:E1::00
     --------

     IP addr  = 192.168.144.109
     ATM addr = 47:00:00:00:00:00:00:00:00:00:CC:BA::08:00:20:84:E3:21::00
     --------

SEE ALSO
     ifconfig(1M), aarconfig(4), ba(7)
```

atmgetmac(1M)

CODE EXAMPLE 6-5 atmgetmac(1M) Man Page

```
atmgetmac(1M)         Maintenance Commands         atmgetmac(1M)



NAME
     atmgetmac - get the MAC address assigned to an ATM interface

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmgetmac interface [count]

AVAILABILITY
     SUNWatm

DESCRIPTION
     atmgetmac retrieves  MAC  addresses  of  the  specified  ATM
     interface  (specified  in the form "device unit;" an example
     is ba0).  If the board has multiple MAC addresses, only  the
     first  one will be returned.  The remaining addresses follow
     sequentially after the first.


OPTIONS
     count          This  flag  requests  the   number   of   MAC
                    addresses  assigned  to  the interface board.
                    SunATM  2.0  boards  have  one  assigned  MAC
                    address, while SunATM 2.1 and 3.0 boards have
                    sixteen assigned MAC addresses.

SEE ALSO
     aarconfig(4), laneconfig(4)
```

atmreg(1M)

CODE EXAMPLE 6-6 atmreg(1M) Man Page

```
atmreg(1M)           Maintenance Commands            atmreg(1M)



NAME
     atmreg - ATM address registration

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmreg  interface   [   -r   |   -d   ]
     atm_address

AVAILABILITY
     SUNWatm

DESCRIPTION
     atmreg communicates with the ILMI daemon, ilmid, which  con-
     trols  notifications to the switch of local address changes.
     The user may register new addresses,  check  the  status  of
     addresses,   or   de-register  addresses.   A  list  of  all
     registered addresses for an interface is printed in the out-
     put of qccstat(1M).

     The first parameter is the  physical  interface  name.  This
     should be specified in the form "device unit;" an example is
     ba0. If neither of the  optional  flags  is  specified,  the
     status  of atm_address is printed. atm_address may be either
     20 or 7 colon-separated hexadecimal octets  (2  characters),
     providing  an entire ATM address or simply the local ESI and
     selector bytes.  If only 7 bytes are provided,  the  default
     13-byte prefix assigned by the switch is assumed.

OPTIONS
     -r   This flag specifies that the given  address  should  be
          registered  on this interface. As soon as the registra-
          tion request has been sent to the switch,  the  program
          will  return;  therefore,  the output of qccstat(1M) or
          atmreg with no flag should be checked  to  verify  that
          the  address  has  been  successfully  registered.  The
          switch will fail an address registration request if the
          same address has already been registered by a different
```

CODE EXAMPLE 6-6 `atmreg(1M)` Man Page *(Continued)*

```
          host.

     -d   This flag specifies that the given  address  should  be
          de-registered  on  this  interface. As is the case with
          the -r flag, the atmreg program will exit  as  soon  as
          the request has been sent to the switch, and successful
          de-registration  should   be   verified   with   either
          qccstat(1M) or atmreg.

EXAMPLES
     The following example shows  three  operations:  first,  the
     status of an address is checked on an interface, which indi-
     cates that the address is not registered. Next, registration
     of the address is requested. Finally, another status request
     is  sent  to  verify  that  the  address  was   successfully
     registered.

     muskogee# atmreg ba0 08:00:20:aa:bb:cc:00
     ATM address
     45:00:00:00:00:00:00:00:0f:00:00:00:00:08:00:20:aa:bb:cc:00
     is unknown on ba0.

     muskogee# atmreg ba0 -r 08:00:20:aa:bb:cc:00
     Requested registration of ATM address on ba0:
     45:00:00:00:00:00:00:00:0f:00:00:00:00:08:00:20:aa:bb:cc:00

     muskogee# atmreg ba0 08:00:20:aa:bb:cc:00
     ATM address
     45:00:00:00:00:00:00:00:0f:00:00:00:00:08:00:20:aa:bb:cc:00
     is registered on ba0.

SEE ALSO
     ilmid(1M), qccstat(1M)
```

atmsetup(1M)

CODE EXAMPLE 6-7 atmsetup(1M) Man Page

```
atmsetup(1M)          Maintenance Commands           atmsetup(1M)



NAME
     atmsetup - configure an ATM device

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmsetup config_file

AVAILABILITY
     SUNWatm

DESCRIPTION
     atmsetup performs ATM configuration, based on  the  informa-
     tion  found in the specifed configuration file.  In general,
     the configuration file should be /etc/atmconfig; the  speci-
     fied  configuration  file  must  have  the  same  format  as
     /etc/atmconfig.

     Configuration of a SunATM device is divided into two phases.
     The  first  consists  of  plumbing all devices, and IP setup
     (using  ifconfig(1M))  for  Classical  IP  interfaces.   The
     second  consists  of  IP setup for LAN Emulation interfaces,
     and is performed by lanesetup(1M).

     atmsetup is called with the appropriate options  during  the
     execution  of  the  SunATM  startup script, S00sunatm, which
     runs during system boot.  Users should not call it from  the
     command prompt.

RETURN VALUES
     On success, atmsetup returns a value indicating the presence
     of  configured  Classical IP interfaces: 0 indicates none, 1
     indicates Classical IP interfaces are present.

     -1 is returned on failure.

SEE ALSO
     ifconfig(1M), aarsetup(1M), lanesetup(1M), atmconfig(4)
```

CODE EXAMPLE 6-7 `atmsetup(1M)` Man Page *(Continued)*

```
NOTES
     Normally,     this     command     is     executed      from
     /etc/rc2.d/S00sunatm.   It  should not be used from the com-
     mand prompt.
```

atmsnmpd(1M)

CODE EXAMPLE 6-8 atmsnmpd(1M) Man Page

```
atmsnmpd(1M)          Maintenance Commands          atmsnmpd(1M)



NAME
     atmsnmpd - ATM SNMP agent daemon

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmsnmpd [ -n ] [ -p port ] [ -f port ]
          [ -t port ] [ -c config-file ] [ -T trace-level ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     The ATM SNMP agent daemon, atmsnmpd, provides a SNMP (Simple
     Network  Management  Protocol)  agent which supports the ATM
     UNI and LAN Emulation Management  Information  Bases  (MIBs)
     defined  in  the  User  Network  Interface and LAN Emulation
     Specifications.  This agent provides information to  a  Net-
     work Management System, such as SunNet Manager.

     Unless otherwise specified, all SNMP  agents  use  the  same
     port  number,  so  a  system  can only support a single SNMP
     agent on a port.  If other SNMP agents are installed on your
     system,  atmsnmpd  must  be  started  with  the -p and/or -f
     options. Alternatively the other agent may be configured  to
     listen on a UDP port other than the default one.  If this is
     not done, atmsnmpd will exit with  an  error  or  cause  the
     other agent to fail.

     If you choose to configure your system as an ATM SNMP  agent
     when  installing  the  SUNWatm package, the software will be
     configured to automatically start  atmsnmpd  at  boot  time.
     Depending  on  the release of Solaris that you're using, the
     port on which the atmsnmpd will be started differs.  Solaris
     2.6  and  above  will  include  a bundled version of an SNMP
     agent that will be started by  default  on  port  161.  This
     means  that  any other agent running on the system will have
     to listen to another UDP port acting  as  a  subagent.  This
```

CODE EXAMPLE 6-8 atmsnmpd(1M) Man Page *(Continued)*

```
     port  can  be  configured by using the atmadmin program, and
     will use a default value  of  1000  for  a  2.6  release  of
     Solaris and above, and a value of 161 otherwise.

     If you choose not to configure your system as  an  ATM  SNMP
     agent,  the software will still start atmsnmpd, but with the
     -n option (see below). This means  that  atmsnmpd  will  not
     listen  for  incoming  requests  on  any  UDP port, but will
     respond to requests coming from ilmid(1M).

     The default configuration information for  the  SunATM  SNMP
     agent  may  be  found  in  the  daemon's configuration file,
     /etc/opt/SUNWatm/snmp/agent.cnf.    Any   changes   to   the
     defaults  may  be  made  in this file; atmsnmpd must be res-
     tarted for any changes to take effect.  In  particular,  the
     default community values are public for read and private for
     write.

OPTIONS
     -p port   Defines an alternative UDP port on which  atmsnmpd
               listens for incoming requests.  The default is UDP
               port 161 for releases of Solaris prior to 2.6,  or
               1000 otherwise.

     -t port   Defines an alternative UDP port on which  atmsnmpd
               sends traps. The default is UDP port 162.

     -f port   Defines a UDP  port  on  which  atmsnmpd  forwards
               unknown  incoming  requests.  If  atmsnmpd  gets a
               response back, it will forward it to the  request-
               ing  SNMP  manager.  The default action is no for-
               warding.

     -n        atmsnmpd will not listen for incoming requests  on
               any  UDP  port  (either the default 161 or the one
               specified with -p). This option  takes  precedence
               over  -p  and is the option with which atmsnmpd is
               started if, during installation, it is not started
               as  an  SNMP  agent. With this option, atmsnmpd is
               used for SNMP requests coming from ilmid(1M).

     -c config-file
               Defines a configuration file that is read when the
               agent  starts  up.  If a configuration file is not
               specified the file /etc/opt/SUNWatm/snmp/agent.cnf
               is used.
```

CODE EXAMPLE 6-8 `atmsnmpd(1M)` Man Page *(Continued)*

```
     -T trace-level
               Sets trace levels.  A value  of  0  disables   all
               tracing   and    is   the   default.    Levels   1
               through 3 represent  increasing  levels  of  trace
               output.  Trace output is sent to the standard out-
               put   in   effect  at   the   time   atmsnmpd   is
               started.

FILES
     /etc/opt/SUNWatm/snmp/agent.cnf(4)
                                 Contains SunATM SNMP agent  con-
                                 figuration information

     /etc/opt/SUNWatm/snmp/acl.cfg(4)
                                 Contains entries for the  access
                                 list control table

     /etc/opt/SUNWatm/snmp/context.cfg(4)
                                 Contains entries for the context
                                 table

     /etc/opt/SUNWatm/snmp/party.cfg(4)
                                 Contains entries for  the  party
                                 table

     /etc/opt/SUNWatm/snmp/view.cfg(4)
                                 Contains entries  for  the  view
                                 table

SEE ALSO
     "ATM User-Network Interface  Specification,  V3.0,  V3.1  or
     V4.0," ATM Forum.

     "LAN Emulation over ATM Specification, V1.0," ATM Forum.

NOTES
     atmsnmpd SHOULD NOT be put into  the  background  (i.e.  run
     with the command 'atmsnmpd &'). When executed, atmsnmpd will
     first perform some essential first steps,  then  put  itself
     into the background without user intervention.
```

atmsnoop(1M)

CODE EXAMPLE 6-9 atmsnoop(1M) Man Page

```
atmsnoop(1M)          Maintenance Commands          atmsnoop(1M)



NAME
     atmsnoop - capture and inspect ATM network packets

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmsnoop [ -aPDSvVNC ] [ -d device ]
          [ -s snaplen ] [ -c maxcount ] [ -i filename ]
          [ -o filename ] [ -n filename ] [ -t [ r | a | d ] ]
          [ -p first [  , last ] ] [ -I vc [ , vc ] [ - vc ] ]
          [ -X vc [ , vc ] [ - vc ] ] [ -x offset [ , length ] ]
          [ -q ] [ expression ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     atmsnoop captures packets from an ATM interface and displays
     their  contents.  The options and functionality are the same
     as the generic snoop command, with a few ATM-specific  addi-
     tions.  The  options that are different from those described
     in the snoop(1M) man page are described  here.  For  a  full
     description  of  the  basic  options,  see the snoop(1M) man
     page.

OPTIONS
     -d device      Receive packets from the  network  using  the
                    interface  specified  by device. If no device
                    is specified using the -d flag, atmsnoop will
                    use ba0 by default.

     -I vc[,vc][-vc]
                    Only display frames from the specified VC(s).
                    A  single  VC, a list of VCs (vc,vc,vc), or a
                    range of VCs (vc-vc) may be specified.   Note
                    that -I 5 directly contradicts the expression
                    nosig; if both of these options appear in the
                    command  line,  an  error will be printed and
```

CODE EXAMPLE 6-9 `atmsnoop(1M)` Man Page *(Continued)*

```
               atmsnoop exits.  The same  is  true  for  the
               combination  of  -I  16  and  the  expression
               noilmi.  However, the combination of -I 5 and
               the  expression noqsaal is allowed; this will
               result in the printing of VC 5 signaling mes-
               sages only.

-X vc[,vc][-vc]
               Do not  display  frames  from  the  specified
               VC(s). A single VC, a list of VCs (vc,vc,vc),
               or a range of VCs (vc-vc) may be specified.

-q             When capturing to a file (-o option)  do  not
               print  a running count of the number of pack-
               ets captured. At high packet  rates  continu-
               ously   printing   the   packet   count  uses
               significant  CPU  time,  the  -q  option  can
               improve atmsnoop's capture performance.

expression     Select packets either  from  the  network  or
               from  a capture file.  Only packets for which
               the expression is true will be selected.   If
               no expression is provided it is assumed to be
               true.

               atmsnoop supports the boolean primitives  and
               operators that are discussed in the snoop(1M)
               man page. In addition, it supports some  atm-
               specific  primitives that may also be used in
               filter expressions. They are:

               nosig
                  When used  as  an  argument  to  atmsnoop,
                  nosig  filters out of the output all pack-
                  ets sent or received on the signalling VC,
                  VC  5,  which  is  used for signalling and
                  QSAAL packets.

               noqsaal
                  QSAAL packets are a subset of  those  seen
                  on  the  signalling  VC.   When noqsaal is
                  used  as  an  argument  to  atmsnoop,   it
                  filters out only the QSAAL packets.

               noilmi
                  ILMI packets (all VC 16 traffic)  will  be
```

CODE EXAMPLE 6-9 `atmsnoop(1M)` Man Page *(Continued)*

```
                       filtered out if noilmi appears as an argu-
                       ment to atmsnoop.

                    nollc
                       The LLC protocol is used to encapsulate IP
                       packets  into  ATM; if the primitive nollc
                       appears as an argument  to  atmsnoop,  all
                       LLC  packets  will  be filtered out of the
                       output. LAN Emulation data frames will  be
                       filtered, since they are LLC encapsulated.

                    nolane
                       All LAN Emulation frames are filtered out.
                       This  includes  both LAN Emulation control
                       frames and data sent  over  LAN  Emulation
                       connections.

EXAMPLES
     Capture all non-ILMI packets on ba0 and display them as they
     are received:

          muskogee# atmsnoop -d ba0 noilmi
          Using device ba0 (promiscuous mode)
          TX: VC=5
          QSAAL: PDU_BGN N(MR)=40 N(UU)=0
          ________________________________________________________
          TX: VC=5
          QSAAL: PDU_BGN N(MR)=40 N(UU)=0
          ________________________________________________________
          TX: VC=5
          QSAAL: PDU_BGN N(MR)=40 N(UU)=0
          ________________________________________________________
          ^Cmuskogee#

     Capture all non-QSAAL packets on ba0  and  save  them  to  a
     file:

          muskogee# atmsnoop -d ba0 -o save noqsaal
          Using device ba0 (promiscuous mode)
          ^Cmuskogee#

     Capture all packets and show the verbose summary output:

          muskogee# atmsnoop -d ba0 -V
          Using device ba0 (promiscuous mode)
          TX: VC=5
```

CODE EXAMPLE 6-9 `atmsnoop(1M)` Man Page *(Continued)*

```
          QSAAL: PDU_POLL N(S)=7 N(PS)=271
          ________________________________________________________
          RX: VC=5
          QSAAL: PDU_STAT N(R)=7 N(MR)=22 N(PS)=271
          ________________________________________________________
          RX: VC=1005
          LLC Type=0x0800 (IP), size = 160 bytes
          IP  D=192.1.1.5 S=192.1.1.8 LEN=148, ID=23478
          UDP D=2049 S=836 LEN=128
          RPC C XID=797246949 PROG=100003 (NFS) VERS=2 PROC=4
          NFS C LOOKUP FH=B609 dir_entry055
          ________________________________________________________
          RX: VC=5
          QSAAL: PDU_POLL N(S)=7 N(PS)=270
          ________________________________________________________
          TX: VC=5
          QSAAL: PDU_STAT N(R)=7 N(MR)=47 N(PS)=270
          ________________________________________________________
          RX: VC=1007
          LLC Type=0x0800 (IP), size = 152 bytes
          IP  D=192.1.1.5 S=192.1.1.12 LEN=140, ID=51245
          UDP D=2049 S=946 LEN=120
          RPC C XID=797034130 PROG=100003 (NFS) VERS=2 PROC=6
          NFS C READ FH=79DA at 0 for 8192
          ________________________________________________________
          ^Cmuskogee#

SEE ALSO
     snoop(1M), ilmid(1M), q93b(7)
```

atmspeed(1M)

CODE EXAMPLE 6-10 atmspeed(1M) Man Page

```
atmspeed(1M)          Maintenance Commands          atmspeed(1M)



NAME
     atmspeed - get and set the total link bandwidth  of  an  ATM
     device

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmspeed interface [ bandwidth ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     atmspeed gets and sets the link bandwidth (wire speed) of an
     ATM  device,  providing  a  mechanism  to  limit  the  total
     bandwidth of the ATM device.  If no bandwidth is  specified,
     the  current  link  bandwidth  is  displayed in Megabits per
     second. If a bandwidth is specified, the link  bandwidth  is
     set to that amount; the total throughput of the link will be
     limited to the  value  specified.  The  specified  bandwidth
     should  be  an  integer  number  of Megabits per second, and
     should be less than the maximum bandwidth that may be  allo-
     cated, which is 135 Mbits/sec in the SunATM-155 products and
     534 Mbits/sec in the SunATM-622 products.  See the ATM  dev-
     ice  man pages (ba(7)) for information on the maximum device
     bandwidth.

EXAMPLES
     The following example shows how the bandwidth of an ATM dev-
     ice  may  be  limited  for a switch that can only handle 100
     Mbits/sec of traffic.  After being  set,  the  bandwidth  is
     checked to verify the correct setting.

          muskogee# atmspeed ba0 100
          muskogee# atmspeed ba0
          100
          muskogee#
```

CODE EXAMPLE 6-10 `atmspeed(1M)` Man Page *(Continued)*

```
SEE ALSO
     ba(7)
```

atmstat(1M)

CODE EXAMPLE 6-11 atmstat(1M) Man Page

```
atmstat(1M)           Maintenance Commands            atmstat(1M)



NAME
     atmstat - display ATM network interface information

SYNOPSIS
     /etc/opt/SUNWatm/bin/atmstat interface [ -d [ -T ] ] [ -t  ]
     [ interval ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     atmstat displays statistics for an ATM  interface.  If  only
     the interface is provided, as shown in the first form of the
     command, a one-line summary for each VC on the ATM interface
     is displayed.  Information is given regarding the mode which
     is being used on each VC, the bandwidth group to which  each
     VC  is  assigned,  and  the  number of incoming and outgoing
     packets for each VC. The interface  parameter  should  be  a
     string of the form baN, where N is the unit number.

     Different output information is provided if one of the flags
     in  the  second or third forms is used. These optional flags
     can be used to display debugging  information  or  bandwidth
     group information.

OPTIONS
     -d        Display debugging information. The output consists
               of  error  and activity counters from the hardware
               device.

     -T        Display timestamp information in addition  to  the
               debugging information provided with the -d option.
               Timestamps are generated by the driver at the time
               the  statistics  are copied from its internal data
               structures. This option  is  useful  to  correlate
               atmstat output with atmsnoop data.
```

CODE EXAMPLE 6-11 `atmstat(1M)` Man Page *(Continued)*

```
     -t        Display the bandwidth group table for  the  inter-
               face.  The bandwidth group table controls the mul-
               tiplexing of packets from multiple  VCs  into  the
               transmit path.

     interval  Display   updated   information   every   interval
               seconds.  The  display  will continue until inter-
               rupted by the user.

EXAMPLES
     The following command displays a summary of  VC  information
     for  ba0 every 5 seconds.  Initially, there are three active
     VCs, the Q.2931 signalling VC 5, the ILMI address  registra-
     tion  VC  16,  and  the  Classical  IP connection to the arp
     server VC 32; during the display, a fourth VC is set up  for
     IP traffic, using Classical IP.

     muskogee# atmstat ba0 5
     ba0  VC   sap  aal  bufsize  ipkts  opkts  encap  BWG  BW(Mb/s)
     ---------------------------------------------------------------
           5   sig    5     9264    492   1233   null    0      0.06
          16  ilmi    5     9264     22     23   null    0      0.06
          32 atmip    5     9264      2      3    llc    4    135.00

     ba0  VC   sap  aal  bufsize  ipkts  opkts  encap  BWG  BW(Mb/s)
     ---------------------------------------------------------------
           5   sig    5     9264    502   1243   null    0      0.06
          16  ilmi    5     9264     23     24   null    0      0.06
          32 atmip    5     9264      2      3    llc    4    135.00

     ba0  VC   sap  aal  bufsize  ipkts  opkts  encap  BWG  BW(Mb/s)
     ---------------------------------------------------------------
           5   sig    5     9264    514   1254   null    0      0.06
          16  ilmi    5     9264     23     24   null    0      0.06
          32 atmip    5     9264      4      6    llc    4    135.00
          33 atmip    5     9264      1      1    llc    4    135.00
     ^C
     muskogee#

     The fields of atmstat's display are:

     VC        The Virtual Circuit to which this line of  statis-
               tics  applies.   The  VC is displayed as a decimal
               number.
```

CODE EXAMPLE 6-11 `atmstat(1M)` Man Page *(Continued)*

```
     sap       The service access point, if any, associated  with
               this  VC.   If the value is for a non-IP data con-
               nection, it is displayed as a hexadecimal  number.
               For  IP  connections,  either  atmip  or  lane  is
               displayed, for Classical IP and LAN Emulation con-
               nections,  respectively.  Utility  VCs used by the
               ATM software are also identified by  name,  rather
               than a numerical service access point.

     aal       The ATM Adaptation Layer used on this VC.

     bufsize   The buffer size, in bytes, being used.

     ipkts     The number of incoming packets received on this VC
               since the VC was established.

     opkts     The number of outgoing packets sent since  the  VC
               was established.

     encap     The type of encapsulation being used.

     BWG       Bandwidth group with which this VC is associated.

     BW(Mb/s)  The total bandwidth (in Mbits per second) which is
               allocated for the BWG associated with this VC.

     The following command displays error and  activity  counters
     for the port ba0:

     muskogee# atmstat ba0 -dT
     timestamp 18:27:28.93043
     intrs              1697143       inits                   2
     ipackets           1817576       opackets            47017
     ierrors                107       oerrors                 0
     out of rbufs             0       out of tbufs            0
     canput fails           107       flow ctls               0
     copy receives      1817576       allocb fails            0
     too many bytes           0       rx overflows            0
     out of txds              0       bad crcs                0
     no receivers             0       err encaps              0
     err acks                 0       txc overflows           0
     rx memnotav              0       rx statenotav           0
     rx badcells              0       rx flush count         65
     rx dirty count           0       rx targ kicks           0
     sbufnum                192       bbufnum                 0
     IP disabled VCs          0       rx bogus len            0
```

CODE EXAMPLE 6-11 atmstat(1M) Man Page *(Continued)*

```
RX PFIFO full           0


The fields of the atmstat -d display are:


intrs     The number of interrupts generated by the device.

inits     The number of times the hardware has been initial-
          ized.

ipackets  The number of packets which have  arrived  on  any
          VCI.

opackets  The number of packets which have been sent on  any
          VCI.

ierrors   The number of input errors.

oerrors   The number of output errors.

out of rbufs
          The number of times the hardware signalled it  had
          to  drop  a  received packet due to no host memory
          buffer. This indicates that packets  are  arriving
          from  the  network faster than the driver can pro-
          cess them.

out of tbufs
          The  number  of  transmitted  packets  which  were
          dropped  because  memory  allocation  failed. This
          indicates  that  the  system  is  running  low  on
          memory.

canput fails
          The number of received packets which were  dropped
          by  the driver because canput() failed. This indi-
          cates that packets are arriving from  the  network
          faster  than software above the driver can process
          them.

flow ctls The number of transmit  packets  which  were  dis-
          carded  because  there  was no transmit descriptor
          available and the software queue was full.

copy receives
```

CODE EXAMPLE 6-11 `atmstat(1M)` Man Page *(Continued)*

```
          The number of received packets  which  were  small
          enough that the driver copied them into a new mblk
          rather than sending up the hardware's  buffer.  It
          is  faster  to copy a small packet than allocate a
          new buffer for the hardware to DVMA  to.  This  is
          not  an  error,  the  counter is for informational
          purposes.

allocb fails
          The number of received packets which were  dropped
          because allocb() failed. This indicates the system
          is running low on memory.

too many bytes
          The number of times the  driver  started  queueing
          transmit  packets  because  there were already too
          many bytes given to the hardware.  "too  many"  is
          defined as 4 Kbytes for every 64 Kbps of requested
          bandwidth for a particular VCI, and  implements  a
          flow  control mechanism to keep low bandwidth con-
          nections from using too much system  memory.  This
          is  not an error, the counter is for informational
          purposes.

rx overflows
          The number of times a received packet was  dropped
          because  it  overflowed  the hardware buffer allo-
          cated for its reception. This generally  indicates
          that  cells  are  being dropped in the ATM network
          due to congestion, causing  cells  from  different
          packets  to  become  concatenated  together into a
          giant packet.

out of txds
          The number of times the  driver  started  queueing
          transmit packets because there were no descriptors
          available on the hardware ring.  This  is  not  an
          error, the counter is for informational purposes.

bad crcs  The number of times a received packet was  dropped
          because its AAL5 CRC was incorrect. This indicates
          a problem in the ATM network.

no receivers
          The number of times a packet arrived on a VCI  for
          which  there was no user. Generally this is a race
```

CODE EXAMPLE 6-11 `atmstat(1M)` Man Page *(Continued)*

```
               condition, the user which allocated that VCI  hav-
               ing  exited  while  packets  were  still in flight
               through the network.

     err encaps
               The number of  recevied  LLC  packets  which  were
               dropped because the indicated SAP had no listener.

     err acks  The number of bus errors which have occurred.  The
               hardware  must be reinitialized when this happens.
               Bus errors can result  from  excessive  electrical
               noise, and indicate a hardware fault.

     txc overflows
               The number of times  the  hardware  indicated  its
               transmit  completion  ring  was full. The hardware
               must be  reinitialized  when  this  happens.  This
               indicates  that  packets are being transmitted way
               faster than the driver can clean up after them, or
               that  the driver was unable to run for an extended
               period of time due to higher  priority  interrupts
               hogging the CPU.

     rx memnotav
               The number of times  the  hardware  indicated  its
               receive  buffer  memory  was  full. This indicates
               that packets are arriving from the network  faster
               than  the  hardware  can  DMA them to host memory.
               This can happen sporadically if other  devices  on
               the  bus  consume  too  much bandwidth for a short
               period of time.

     rx statenotav
               The number of times  the  hardware  indicated  its
               receive  control  memory  was full. This indicates
               that packets are arriving from the network  faster
               than the hardware can process them.

     rx badcells
               The number of cells which arrived to the  hardware
               destined  for  a VCI which was not turned on. This
               often happens  with  switches  configured  to  use
               SPANS  signalling,  which  sends  cells  to VCI 15
               looking for a SPANS-capable device.

     rx flush count
```

CODE EXAMPLE 6-11 `atmstat(1M)` Man Page *(Continued)*

```
              The number of DMA states loaded into the  RX  con-
              trol  memory. This is not an error, the counter is
              for informational purposes.

    rx dirty count
              The number of DMA states loaded into the  RX  con-
              trol  memory  when there was no clean state avail-
              able. The hardware has to flush one of the  exist-
              ing  states  to  external RAM. This indicates that
              the hardware is approaching  its  limits  for  the
              number of simultaneously active VCIs, but is still
              able to keep up. This is not an error, the counter
              is for informational purposes.

    rx targ kicks
              The number of times the driver had to instruct the
              hardware  to  move  its  targeted channels back to
              their private buffer rings.  This  indicates  that
              either the incoming traffic load is truly monumen-
              tal, or that the driver was unable to run  for  an
              extended period due to a higher priority interrupt
              hogging the CPU. This is not an error, the counter
              is for informational purposes.

    sbufnum   The number of buffers available to the hardware on
              the  non-targeted  buffer  ring. This ring is used
              for VCIs requesting the small or big buffer  size.
              This  is not an error, the counter is for informa-
              tional purposes.

    bbufnum   The number of buffers available to the hardware on
              the non-IP buffer ring. This ring is used for VCIs
              requesting the huge buffer size. Buffers for  this
              ring  are not allocated by the driver until a user
              requests the huge buffer  ring.  This  is  not  an
              error, the counter is for informational purposes.

    IP disabled VCs
              The number of packets sent from the IP  stream  to
              VCIs the driver thinks are turned off. If the q93b
              link has recently gone down this is normal (a sim-
              ple  race  condition between IP and the driver). A
              large number of these errors would indicate a sig-
              nalling problem.

    rx bogus len
```

CODE EXAMPLE 6-11 `atmstat(1M)` Man Page *(Continued)*

```
               The number of times a received packet was  dropped
               because  its claimed AAL5 length did not match the
               number of cells received  by  the  hardware.  This
               indicates  a problem with some piece of ATM equip-
               ment sending cells to the adaptor;  in  particular
               misconfigured ATM analyzers can do this.

     rx PFIFO full
               The number of times a received packet was  dropped
               because  the  queue  used  by the software to send
               them up to higher protocol layers was  full.  This
               indicates  that  there are so many hardware inter-
               rupts generated by devices in the system that  the
               software interrupt is never able to run.


SEE ALSO
     ifconfig(1M), netstat(1M), ba(7)
```

ilmid(1M)

CODE EXAMPLE 6-12 ilmid(1M) Man Page

```
ilmid(1M)            Maintenance Commands             ilmid(1M)



NAME
     ilmid - ATM Address Registration daemon

SYNOPSIS
     /etc/opt/SUNWatm/bin/ilmid [ -c ] [ -n ] [ -v ] [ -x ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     The ATM Address Registration daemon  communicates  with  the
     switch to establish the 20-byte ATM address for the end sys-
     tem.  It implements ILMI, which is the Interim Local Manage-
     ment  Interface specified in the ATM User Network Specifica-
     tion.  It uses the Simple Network Management Protocol (SNMP)
     for communication between an ATM switch and host.

     An ATM address is made up of a 13-byte network prefix, a  6-
     byte  end  system  identifier  (esi), and a 1-byte selector.
     Currently, the selector byte  is  not  used  in  the  SunATM
     implementation;  it  will  be 00 in most cases.  The network
     prefix is assigned by the switch and will  be  used  by  the
     switch for routing.  The esi is the unique identification of
     the end system.  A good choice for this is often the default
     MAC  address  for  the interface.  For all Sun products, the
     MAC address will begin with the octets 08:00:20.

     When the ilmi daemon is executed,  it  first  registers  the
     local  MAC address for each interface, obtained from the ATM
     driver, with the switch. Part of  the  initial  registration
     process  involves  obtaining  the switch prefix, which ilmid
     reports to the ATM software.  It then waits to receive  mes-
     sages  from  user  programs  or  the switch, and responds to
     those accordingly.

     Additional addresses may  be  registered  in  two  different
```

CODE EXAMPLE 6-12 `ilmid(1M)` Man Page *(Continued)*

```
     ways.  aarsetup(1M)  and  lanesetup(1M)  register additional
     addresses that may appear in aarconfig(4) and laneconfig(4),
     respectively.   There  is  also  a user program, atmreg(1M),
     that may be used to register and de-register addresses,  and
     also check the status of any address.

OPTIONS
     -c     Clear address table. Normally, when ilmid is started,
            it  obtains  a list of all addresses that were previ-
            ously registered  from  the  ATM  software,  and  re-
            registers  all of them. Using the -c option instructs
            ilmid instead to only register  the  default  address
            for  each  interface,  and  clear all other addresses
            from the ATM software address table.

     -n     No auto registration. By default, ilmid automatically
            registers  a  local address with the switch, which is
            made  up  of  the  switch  prefix,  the  MAC  address
            assigned  to  the  board (or system if the board does
            not have its own), and  a  0  selector.  This  option
            turns  off  that  feature, so that the only addresses
            registered are those that appear in  'l'  entries  in
            /etc/aarconfig and/or /etc/laneconfig.

     -v     Verbose mode. Print additional information  regarding
            the communication with the switch.

     -x     Print (to the console) the messages exchanged between
            the switch and end system in hexadecimal notation.

SEE ALSO
     atmreg(1M),   aarsetup(1M),   lanesetup(1M),   aarconfig(4),
     laneconfig(4)

     "ATM User-Network Interface Specification, V3.0," ATM Forum.

     "ATM User-Network Interface Specification, V3.1," ATM Forum.

NOTES
     ilmid SHOULD NOT be put into the background (i.e.  run  with
     the command 'ilmid &'). When executed, ilmid will first per-
     form some essential first steps, then put  itself  into  the
     background  without user intervention.  An exception is made
     if ilmid is run with debug flags (-x and/or -v); since those
```

CODE EXAMPLE 6-12 `ilmid(1M)` Man Page *(Continued)*

```
     modes result in continuous output, ilmid will not put itself
     into the background if running with the -x or -v option.
```

lanearp(1M)

CODE EXAMPLE 6-13 lanearp(1M) Man Page

```
lanearp(1M)           Maintenance Commands           lanearp(1M)



NAME
     lanearp - MAC to ATM address resolution

SYNOPSIS
     /etc/opt/SUNWatm/bin/lanearp laneN

     /etc/opt/SUNWatm/bin/lanearp laneN [ MAC address ]

     /etc/opt/SUNWatm/bin/lanearp laneN - [ ATM address ]

     /etc/opt/SUNWatm/bin/lanearp -a

AVAILABILITY
     SUNWatm

DESCRIPTION
     The lanearp program may be  used  to  display  ATM  and  MAC
     address  pairs  for  a  given  LAN Emulation interface.  The
     required parameter laneN is a LAN Emulation  inteface  name,
     where  N  is the LAN Emulation instance number (specified in
     /etc/atmconfig). An example is lane0.

     If only the interface is provided, as in the first  form  of
     the  command,  lanearp  will  print  the ATM address and MAC
     address for that LAN Emulation interface, and an  entry  for
     each resolved IP address for that interface.

     If additional information is provided, it will  be  used  to
     identify  a  device on the subnet to which the LAN Emulation
     interface is connected, and the corresponding address infor-
     mation  will  be  printed.   In the second form, when an MAC
     address (in the colon-separated form used in the  output  of
     arp)  is  provided,  the  ATM  address for that node will be
     printed. In the third form, when  an  ATM  address  (in  the
     colon-separated  octet  format  used  in /etc/laneconfig) is
     provided, the corresponding MAC  address  will  be  printed.
```

CODE EXAMPLE 6-13 `lanearp(1M)` Man Page *(Continued)*

```
     Note:   in this third form of the command, a hyphen (-) must
     be included to indicate that a MAC address is not being pro-
     vided.

     The -a option dumps the complete LANE ARP table, listing the
     ATM  and  IP  address for each LAN Emulation interface and a
     listing of the ATM address for each resolved IP  address  on
     that interface.


EXAMPLES
     sunatm1# lanearp -a
     LANE Interface lane2:
     Local MAC addr = 8:0:20:82:4f:f6
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::08:00:20:82:4F:F6::00

     LE_ARP table:
     -------------
     MAC addr = 0:e0:f9:c5:58:0
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:CC:00::00:E0:F9:C5:58:00::36
     --------
     MAC addr = 8:0:20:7e:58:6
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::08:00:20:7E:58:06::00
     --------
     MAC addr = 8:0:20:82:4f:f6
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::08:00:20:82:4F:F6::00
     --------
     MAC addr = ff:ff:ff:ff:ff:ff
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::00:60:47:2C:3E:04::36
     --------



     LANE Interface lane1:
     Local MAC addr = 8:0:20:82:4f:f5
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::08:00:20:82:4F:F5::00

     LE_ARP table:
     -------------
     MAC addr = 0:e0:f9:c5:58:0
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:CC:00::00:E0:F9:C5:58:00::35
     --------
     MAC addr = 8:0:20:7e:58:5
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::08:00:20:7E:58:05::00
     --------
     MAC addr = 8:0:20:82:4f:f5
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::08:00:20:82:4F:F5::00
```

CODE EXAMPLE 6-13 `lanearp(1M)` Man Page *(Continued)*

```
     --------
     MAC addr = ff:ff:ff:ff:ff:ff
     ATM addr = 47:00:00:00:00:00:00:00:00:00:00:C0:01::00:60:47:2C:3E:04::35
     --------



SEE ALSO
     arp(1M), ifconfig(1M), laneconfig(4),
```

lanesetup(1M)

CODE EXAMPLE 6-14 lanesetup(1M) Man Page

```
lanesetup(1M)        Maintenance Commands         lanesetup(1M)



NAME
     lanesetup - LAN Emulation setup program

SYNOPSIS
     /etc/opt/SUNWatm/bin/lanesetup [ -pnvf ] [ -a filename  ]  [
     filename ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     The lanesetup program reads local LAN  Emulation  configura-
     tion information from the /etc/laneconfig file and loads the
     information into the kernel.

     By default, the /etc/laneconfig file is read and  downloaded
     into  the  local kernel table on startup.  If the configura-
     tion file is modified later, lanesetup must be rerun to load
     the new information into the kernel.

OPTIONS
     -p         Prints to the standard output the  table  entries
               from  the  configuration  file,  with all variable
               expressions expanded. Does not download any infor-
               mation into the kernel.

     -n         Only parse the configuration table.   Using  this
               option,  the  syntax  and information in the table
               can be checked to verify that it is acceptable  to
               the  lanesetup program without actually attempting
               to download any data.  Physical interface informa-
               tion  entered  in the table is compared with known
               configured interfaces.   Error  messages  will  be
               printed if any problems are encountered.

     -v         Verbose mode.  Additional information is printed.
```

CODE EXAMPLE 6-14 lanesetup(1M) Man Page *(Continued)*

```
     -f         Also do the  LAN  Emulation  interface  plumbing.
               This  is  only  done once, at boot time.  The only
               time this option should be used is when  lanesetup
               is  called  in  the  /etc/rc2.d/S00sunatm  startup
               script.

     -a filename
               Used in conjunction with the -f option, this  flag
               specifies the file from which plumbing information
               should be read (typically /etc/atmconfig).

     filename  A filename may be specified to download  a  confi-
               guration  file  other than /etc/laneconfig.  Stan-
               dard input, indicated with  a  hyphen  `-',  is  a
               legal value for filename if the -n option is being
               used.

FILES
     /etc/laneconfig     File that contains configuration  infor-
                         mation  specific  to  the  LAN Emulation
                         interfaces.  Read  by  lanesetup,  which
                         downloads  the configuration information
                         to the LAN Emulation kernel software.

SEE ALSO
     laneconfig(4)

     ATM Forum, LAN Emulation Over ATM Specification Version 1.0,
     LAN Emulation SWG Drafting Group

NOTES
     lanesetup SHOULD NOT be put into the  background  (i.e.  run
     with  the  command  'lanesetup &'). When executed, lanesetup
     will first perform some  essential  first  steps,  then  put
     itself into the background without user intervention.
```

lanestat(1M)

CODE EXAMPLE 6-15 lanestat(1M) Man Page

```
lanestat(1M)         Maintenance Commands          lanestat(1M)



NAME
     lanestat - display status of LAN Emulation over ATM

SYNOPSIS
     /etc/opt/SUNWatm/bin/lanestat lane_interface

     /etc/opt/SUNWatm/bin/lanestat -a

AVAILABILITY
     SUNWatm

DESCRIPTION
     lanestat displays information about the  state  of  the  LAN
     Emulation  protocol  on  an  ATM interface.  The information
     provided may be used to debug configuration problems, or  to
     verify successful bring-up of a LAN Emulation interface.

     The only parameter is  the  LAN  Emulation  interface  name,
     which  will  be  of  the form laneN, where N is the instance
     number. Optionally, the -a  flag  may  be  used  to  request
     information for all LAN Emulation interfaces.

     The following fields will be displayed:

     setup_state    The state of the LAN Emulation setup program,
                    lanesetup.  The  possible  values  are setup-
                    not-run, which means that lanesetup  has  not
                    been  run  successfully  for  this interface;
                    setup-started, which means that lanesetup  is
                    currently    running;   setup-requested-join,
                    which means that a join request has been sent
                    to  the  LES, but a response has not yet been
                    received; setup-finished,  which  means  that
                    lanesetup  has  successfully  completed;  and
                    interface-defunct,  which  means   that   the
                    interface  has been partially unconfigured by
```

CODE EXAMPLE 6-15 `lanestat(1M)` Man Page *(Continued)*

```
                   removing its entries from  the  configuration
                   files  and re-running laneconfig.  Interfaces
                   whose  state  is  interface-defunct  will  be
                   removed  from  the kernel on reboot, assuming
                   that the configuration files are not changed.

     arpcsmode     The mode in which the LAN Emulation  software
                   is  running.  The possible values are client-
                   being-modified  and   client.   client-being-
                   modified   indicates   that   lanesetup   is
                   currently running on the system;  the  confi-
                   guration  is  not  complete; client indicates
                   that the system is a LAN Emulation client.

     proto_address The protocol address of this lane instance.

     atm_address   The ATM address of this lane instance.

     lanestate     The state of the LAN Emulation client. When a
                   LAN  Emulation  client interface comes up, it
                   must go through a process called "joining the
                   LAN."  The  value  in this field reflects the
                   current  stage  in  that  process.   For   a
                   description   of  the  steps  a  client  goes
                   through to join a LAN, see section 5.3.1, LAN
                   Emulation Services, in the SunATM 2.1 Manual.
                   For a client that  is  up  and  running,  the
                   value of this field should be active.

     lecConfigSource
                   The source of the LECS address used  to  con-
                   figure  this  lane  instance.   The  possible
                   values are LocalInformation, which means  the
                   address  is  provided  in the laneconfig file
                   using the 'c' flag; getAddressViaIlmi,  which
                   means  the address was provided by the switch
                   via the  ILMI  daemon;  usedWellKnownAddress,
                   which  means the well-known LECS address from
                   the  ATM  Forum  UNI   standard   was   used;
                   usedLecsPvc,  which  means  the  LECS VCI was
                   provided in  /etc/laneconfig  using  the  'c'
                   flag;  and didNotUseLecs, which means the LES
                   address was provided in /etc/laneconfig using
                   the 's' flag.

     driver name   The ATM hardware device  this  lane  instance
```

CODE EXAMPLE 6-15 `lanestat(1M)` Man Page *(Continued)*

```
                   runs over.

    lan_type       The type of Emulated  LAN.   Possible  values
                   are   unspecified,   ethernet(802.3),  token-
                   ring(802.5), and <unknown>. Currently, SunATM
                   supports  only  emulated  LANs of type ether-
                   net(802.3).

    elan_name      The name of the Emulated LAN.  Most LAN  Emu-
                   lation  Servers will provide this information
                   to the client when the client joins the  LAN,
                   but  in  some  cases,  such as in the case of
                   multiple Emulated LANs, the user must provide
                   this name in its requests to join. If this is
                   the  case  in  your  configuration,  see  the
                   description of the `n' flag in laneconfig(4).

    lecid          A number assigned  by  the  LES  to  uniquely
                   identify this LAN Emulation client.

    max_frame_size_code
    size           A code identifying the maximum SDU size of an
                   Emulated  LAN  data  frame;  the  actual size
                   corresponding to  the  code  is  provided  as
                   well.   This value is generally determined by
                   the LAN Emulation Configuration Server.

    LECS_atm_address
                   The atm address of the  LECS  for  this  lane
                   instance.

    LES_atm_address
                   The atm address of  the  LES  for  this  lane
                   instance.

    BUS_atm_address
                   The atm address of  the  BUS  for  this  lane
                   instance.

    lecs_vci

    les_vci

    les_distribute_vci

    bus_vci
```

CODE EXAMPLE 6-15 `lanestat(1M)` Man Page *(Continued)*

```
     bus_forward_vci
                    The VCIs identifying the connections  to  the
                    three  servers  providing  LAN Emulation ser-
                    vices for the emulated LAN.  A VCI of 0 indi-
                    cates  that no connection exists.  It is nor-
                    mal for the LECS connection to be  torn  down
                    during  the  process  of joining the emulated
                    LAN.

SEE ALSO
     lanesetup(1M), laneconfig(4)
```

qccstat(1M)

CODE EXAMPLE 6-16 qccstat(1M) Man Page

```
qccstat(1M)           Maintenance Commands            qccstat(1M)



NAME
     qccstat - display Q.2931 call control information

SYNOPSIS
     /etc/opt/SUNWatm/bin/qccstat interface [ interval ]

AVAILABILITY
     SUNWatm

DESCRIPTION
     qccstat displays signalling and link layer  information  for
     an ATM interface.  The information includes the current link
     state, ATM addresses registered for the interface,  and  the
     state of all Q.2931 calls present on the interface.

     Without options, qccstat displays several lines of  informa-
     tion  for  the specified interface.  The interface parameter
     is a string of the form baN, where N is the unit number.  If
     interval  is  given,  the  information  will  be updated and
     printed every interval seconds, repeating until  interrupted
     by the user.

     If there are no calls present on the interface, several sum-
     mary lines are displayed.  They include the following infor-
     mation:

          linkstate   The  DLPI  link   state,   usually   either
                      DL_ACTIVE or DL_IDLE.

          outcalls    The total number of outgoing calls on  this
                      interface.

          incalls     The total number of incoming calls.

          sig         The signalling version that is  plumbed  on
                      this interface. Possible values are UNI3.0,
```

CODE EXAMPLE 6-16 qccstat(1M) Man Page *(Continued)*

```
                      UNI3.1, and UNI4.0.

          registered addresses
                      A list of  the  addresses  that  have  been
                      registered  for  this  interface  with  the
                      switch.

     If calls are present, three additional lines of  information
     are provided for each call.  These lines include the follow-
     ing information:

          callref     The call reference for this call.

          vci         The virtual circuit  identifier,  displayed
                      in decimal.

          state       The Q.2931 call state.

          dir         The direction of the call.  If this  system
                      initiated the call, the direction is OUTGO-
                      ING; otherwise, the direction is INCOMING.

          sap         The service access point  (sap)  associated
                      with  this call, displayed as a hexadecimal
                      number.

          src         The 20-byte  source  ATM  address  for  the
                      call,  in  the colon-separated octet format
                      used in the aarconfig(4) file.

          dst         The destination ATM address for the call.

EXAMPLES
     The following command displays Q.2931 call  information  for
     ba0  every 2 seconds.  Initially, there are no active calls;
     during the display, a call is  connected.   The  display  is
     then terminated by the user.

     muskogee# qccstat ba0 2
     ba0: linkstate=DL_ACTIVE outcalls=0 incalls=0 sig=UNI3.0
          registered addresses:
               45:00:00:00:00:00:00:0f:00:00:00:00::08:00:20:75:a2:77:00

     ba0: linkstate=DL_ACTIVE outcalls=0 incalls=0 sig=UNI3.0
          registered addresses:
               45:00:00:00:00:00:00:0f:00:00:00:00::08:00:20:75:a2:77:00
```

CODE EXAMPLE 6-16 `qccstat(1M)` Man Page *(Continued)*

```
     ba0: linkstate=DL_ACTIVE outcalls=0 incalls=1 sig=UNI3.0
          registered addresses:
               45:00:00:00:00:00:00:00:0f:00:00:00:00::08:00:20:75:a2:77:00

          incoming calls:
             callref=1 vci=0x20 state=ACTIVE dir=INCOMING sap=0x800
             src=47:00:05:80:ff:e1:00:00:00:f1:24:0e:e8::08:00:20:10:0a:2d:00
             dst=47:00:05:80:ff:e1:00:00:00:f1:24:0e:e8::08:00:20:22:21:b1:00

     ^Cmuskogee#

SEE ALSO
     q93b(7), ba(7)
```

Index