

Application Programmers' Interface



The Application Programmers' Interface (API) that is provided with this software release is an interim API to be used until the ATM Forum standardizes an API.

Note – Be aware that since this is an interim API, it can be changed at any time.

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

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.

An additional set of functions is provided to facilitate communication with the ATM device driver (the `ba` driver in SunATM software). These functions are referred to collectively as the `atm_util` functions.

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

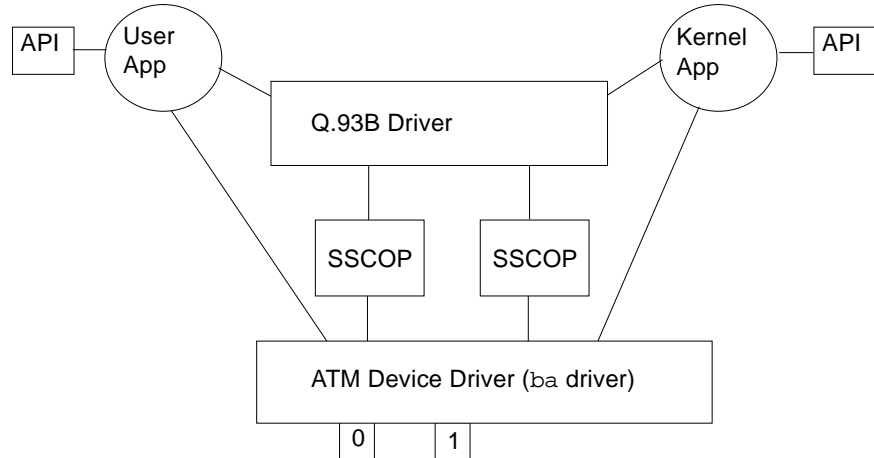


Figure E-1 ATM Signalling

E.1 Using the SunATM API with the q93b and the ATM Device Drivers

The architecture illustrated in Figure E-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.

E.1.1 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).

E.1.2 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 E-1.

Table E-1 Messages Between the User and the q93b Driver

Message Type	Direction
SETUP	BOTH
SETUP_ACK	UP
CALL_PROCEEDING	BOTH
CONNECT	BOTH
CONNECT_ACK	UP
RELEASE	DOWN
RELEASE_COMPLETE	BOTH
STATUS_ENQUIRY	DOWN
STATUS	UP
RESTART	BOTH
RESTART_ACK	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 may be plumbed above the driver, and multiple physical interfaces may be connected below q93b. Applications may 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 E-2; kernel applications should use `put(9f)` to send the mblocks shown, and user applications should send two corresponding strbufs using `putmsg(2)`.

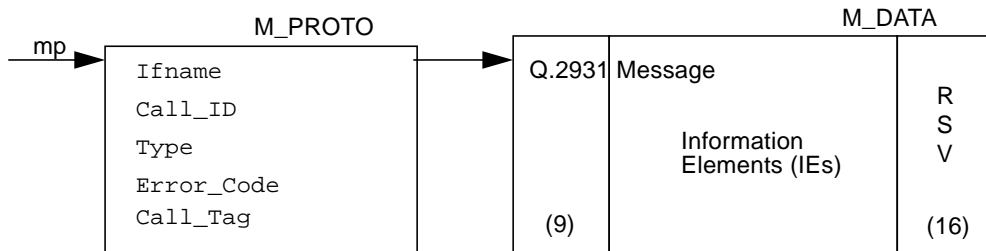


Figure E-2 Message Format

Table E-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 may be used to identify the 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 `<atm/qccotypes.h>`. In the second mblock, the application shall 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 may 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 or 3.1. For further information on the qcc functions, which are outlined in Table E-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 typing: `man qcc_bld`, `man qcc_bld`, `man qcc_bld_setup`, or `man qcc_bld_connect`, etc. The message flow during typical call setup and tear down is diagrammed in Figure E-3 on page E-10.

Table E-3 qcc Functions

Name	Functionality	Input	Output
<code>qcc_bld_*</code>	Creates and encodes a message; allows 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 E-2)
<code>qcc_parse_*</code>	Extracts a defined set of values from an encoded message	Encoded Q.2931 message (in the format shown in Figure E-2)	Parameter values
<code>qcc_len_*</code>	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 <code>qcc_bld/qcc_pack</code> functions.	none	Maximum length of the message.
<code>qcc_create_*</code>	Create a message structure with the required values set. The structure may then be further customized using <code>qcc_set_ie</code> .	Default parameter values	Message structure (defined in <code><atm/qcctypes.h></code>)
<code>qcc_set_ie</code>	Updates or inserts values for an information element into a message structure.	Message structure and IE structure (defined in <code><atm/qcctypes.h></code>)	Updated message structure

Table E-3 qcc Functions (Continued)

Name	Functionality	Input	Output
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 E-2.	Message structure (defined in <atm/qcctypes.h>)	Encoded Q.2931 message (in the format shown in Figure E-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 E-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

E.1.2.1 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, CONNECT, or RELEASE_COMPLETE message from q93b (all other messages are ignored by q93b). After the CONNECT message is received, the user may use the virtual channel.

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

E.1.2.2 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 shall release 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 shall send a RELEASE message to q93b.

E.1.2.3 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 section 5.4.5.15 of the ATM Forum UNI standard.

E.1.3 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 E-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 E-4 `atm_util` Function Overview

Name	Functionality	Kernel Equivalent
<code>atm_open</code>	Open a stream to the ATM device driver	must be done by a user space setup program
<code>atm_close</code>	Close a stream to the ATM device driver	must be done by a user space setup program
<code>atm_attach</code>	Attach to a physical interface	must be done by a user space setup program
<code>atm_detach</code>	Detach from a physical interface	must be done by a user space setup program
<code>atm_bind</code>	Bind to a Service Access Point	send DL_BIND_REQ
<code>atm_unbind</code>	Unbind from a Service Access Point	send DL_UNBIND_REQ

Table E-4 atm_util Function Overview (Continued)

Name	Functionality	Kernel Equivalent
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 E-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.

E.1.3.1 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 four-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 four-byte `vpci` from the message if the application has not set `DLIOCRAW` with a call to `atm_setraw`; if `DLIOCRAW` has been set, the four-byte `vpci` will be included in the message sent up to the application.

E.1.3.2 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.

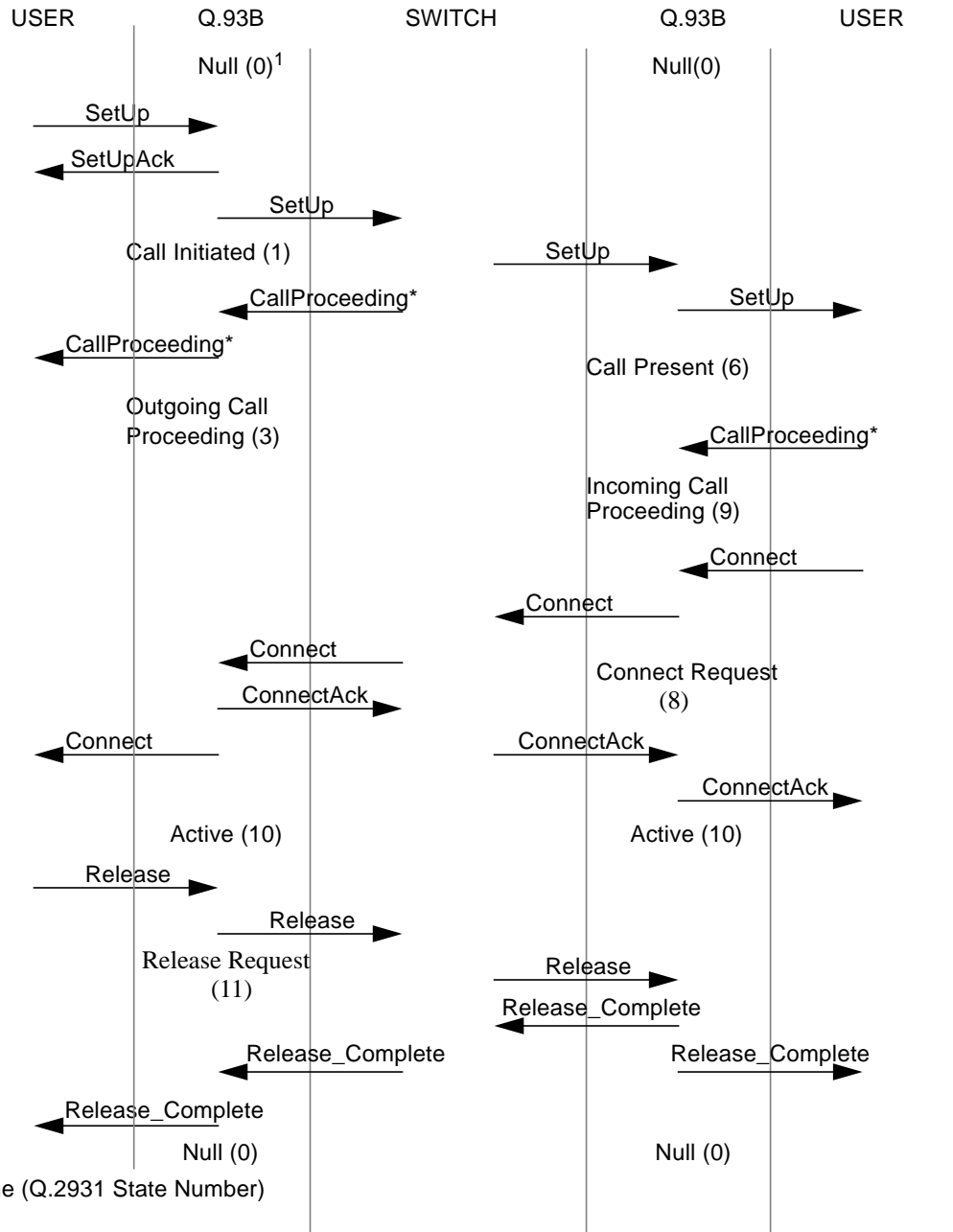


Figure E-3 Normal Call Set up and Tear Down

¹ XX(n): Q.2931 State Name (Q.2931 State Number)
 * Optional