NAME

atm_accept − accept a pending connection request

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

atm_accept(fd, fdnew, conn_id, qos, dataflow)
int fd;
int fdnew;
int conn_id;
Atm_qos *qos;
Atm_dataflow dataflow;

DESCRIPTION

atm_accept( ) is used to accept a connection request and optionally connect back to the remote ATM end-point. The *fd* argument specifies the file descriptor that was used with **atm_listen**(4N). *fdnew* specifies the file descriptor to accept the connection on. *fdnew* then becomes a new local ATM end-point to be used with **atm_send**(4N) and **atm_recv**(4N).

fd and *fdnew* can differ or be equal. When equal, then the connection is accepted and the file descriptor goes from a listening state to a connected state. New connection requests can no longer be accepted using the file descriptor.

When the file descriptors differ, then the file descriptor specified by *fd* remains in a listening state. *fd* can continue to be used with **atm_listen**(4N), while *fdnew* becomes a new local ATM end-point.

Multiple connect requests can arrive over the file descriptor identified by *fd.* Connect requests can be accepted in any order. The *conn_id* argument returned by **atm_listen(** is used to match connection requests with acceptance.

If the caller wishes to connect back to the remote ATM end-point then the *qos* (Quality Of Service) argument specifies the caller's requested target and minimum acceptable network bandwidth requirements for that connection. The ATM network uses this information to allocate network bandwidth during connection establishment.

The *dataflow* argument specifies simplex, duplex, or multicast communication. When simplex communication is selected, the connection is simply accepted, resulting in a dataflow from the remote ATM end-point to the local ATM end-point. If duplex is requested then the connection is accepted and a connection is made back to the remote ATM end-point. If multicast is specified then the connection is accepted as a branch of a unidirectional multicast tree rooted at the remote ATM end-point. Applications must agree on simplex, duplex, or multicast communication beforehand.

See **atm_server**(4N) for usage of **atm_accept( ).**

RETURN VALUES

atm_accept( ) returns 0 on success. On failure, it returns -1 and sets **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors returned by **atm_accept( ).**

ERRORS

DL_OUTSTATE
: No outstanding connection requests.

DL_UNSUPPORTED
: Dataflow not simplex or duplex.

DL_BADCORR
: Bad correlation number. *conn_id* does not match any returned by **atm_listen(.**

ECONNREFUSED
: Connection failed because the destination ATM end-point did not respond. Will also fail if the minimum requested bandwidth could not be allocated in the reverse direction.

atm_accept( ) can also fail with errors returned from the **getmsg**(2) and **putmsg**(2) system calls on the

SunOS and IRIX platforms.

On ULTRIX platforms, **atm_accept( )** on a closed connection may cause a SIGPIPE signal to be sent to the calling process. Unless caught or ignored, this terminates the process.

NOTES

A concurrent server model is not supported in this software release. *fd* and *fdnew* must be equal. If the files descriptors are not equal then error DL_UNSUPPORTED is returned to the caller.

SEE ALSO

atm_intro(4N)

NAME

atm_bind − bind an ASAP to an ATM end-point

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

int atm_bind(fd, asap, asap_assigned, qlen)
int fd;
Atm_sap asap;
Atm_sap *asap_assigned;
int qlen;

DESCRIPTION

atm_bind( ) associates an Application Service Access Point (ASAP) with an open file descriptor specified by *fd.*

The *asap* argument specifies which ASAP to bind to the file descriptor. A value of zero can be specified in which case the ATM device driver will allocate an ASAP and bind it to the file descriptor. ASAP values in the range 0 to SAPS_RESERVED are reserved by the ATM device driver for kernel use.

The *asap_assigned* argument is returned to the caller and specifies the ASAP that was bound to the file descriptor.

The *qlen* argument specifies whether the caller will be a server or a client. A value of zero indicates a client. A non-zero value indicates server; in this case the file descriptor is activated for incoming connection requests with the maximum number of outstanding connection requests limited by the value of *qlen.*

See **atm_client**(4N) and **atm_server**(4N) for usage of **atm_bind( ).**

RETURN VALUES

atm_bind( ) returns 0 on success. On failure, it returns -1 and sets **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors returned by **atm_bind( ).**

ERRORS

DL_OUTSTATE	File descriptor in a state other than DL_UNBOUND.
DL_BOUND	The specified *asap* is already in use.
DL_BADSAP	The specified *asap* is a reserved ASAP.
ENOBUFS	Memory allocation failed.

atm_bind( ) can also fail with errors returned from the **getmsg**(2) and **putmsg**(2) system calls on the SunOS and IRIX platforms.

On ULTRIX platforms, **atm_bind( )** on a closed connection may cause a SIGPIPE signal to be sent to the calling process. Unless caught or ignored, this terminates the process.

SEE ALSO

atm_intro(4N)

NAME

atm_client_aal4 − example client ATM program using AAL 3/4

EXAMPLE

```
/*
 * Copyright (c) 1992 by Fore Systems, Inc.
 */

#ifndef lint
static char _atm_client_aal4_c_[] = "@(#)$Id: atm_client_aal4.c,v 1.3 1994/06/28 20:18:37 jshirron Exp $ FSI";
#endif /* lint */

#include <stdio.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <fore_atm/fore_atm_user.h>

#define SERVER_SAP      4096
#define PROGRESS        1024     /* report progress every n packets */

main(argc, argv)                                                      main
    int         argc;
    char        *argv[];
{
    int fd, i, j, mtu, qlen;
    Atm_info info;
    Atm_endpoint dst;
    Atm_qos qos;
    Atm_qos_sel qos_selected;
    Atm_sap ssap;
    Aal_type aal = aal_type_4;
    Atm_dataflow dataflow = simplex;
    char *device_name;
    char *tbuf, *rbuf;
    extern char *malloc();

    if (argc < 3) {
        fprintf(stderr, "Usage: %s device server−hostname [server−sap]\n",
                    argv[0]);
        exit(1);
    }

    device_name = malloc(strlen("/dev/") + strlen(argv[1]) + 1);
    sprintf(device_name, "/dev/%s", argv[1]);

    if ((fd = atm_open(device_name, O_RDWR, &info)) < 0) {
        perror("atm_open");
        exit(1);
    }
    mtu = info.mtu; /* Max packet size */
    tbuf = malloc(mtu);
    rbuf = malloc(mtu);

    /*
     * Let ATM driver assign a source SAP by specifying
     * sap of zero. Set incoming connect request length to
     * zero since we´re the client. Servers set queue length
     * to non zero.
     */
    ssap = 0;
    qlen = 0;
    if (atm_bind(fd, ssap, &ssap, qlen) < 0) {
        atm_error("atm_bind");
        exit(1);
    }
    printf("SAP assigned=%d\n", ssap);
```

```
        /*
         * Initialize the server´s data link address.
         */
        if (atm_gethostbyname(argv[2], &dst.nsap) < 0) {
            fprintf(stderr, "atm_gethostbyname failed\n");
            exit(1);
        }
        dst.asap = (argc == 4) ? atoi(argv[3]) : SERVER_SAP;

        /*
         * Request some quality of service.
         */
        qos.peak_bandwidth.target = 0; /* kbit/sec */
        qos.peak_bandwidth.minimum = 0; /* kbit/sec */
        qos.mean_bandwidth.target = 128; /* kbit/sec */
        qos.mean_bandwidth.minimum = 64; /* kbit/sec */
        qos.mean_burst.target = 2;   /* 2 kbit packet length */
        qos.mean_burst.minimum = 1; /* 1 kbit packet length */

        if (atm_connect(fd, &dst, &qos, &qos_selected, aal, dataflow) < 0) {
            atm_error("atm_connect");
            exit(1);
        }
        printf("selected qos peak=%d, mean=%d, burst=%d\n",
            qos_selected.peak_bandwidth,
            qos_selected.mean_bandwidth,
            qos_selected.mean_burst);

        for (i = 1; i <= mtu; i++) {

            if (atm_send(fd, tbuf, i) < 0) {
                atm_error("atm_send");
                exit(1);
            }

            switch (dataflow) {
            case simplex:

                if ((i % PROGRESS) == 0)
                    printf("%d packets sent\n", i);

                   for (j = 0; j < i * 3; j++)
                          ;
                break;

            case duplex:

                if (atm_recv(fd, rbuf, i) < 0) {
                    atm_error("atm_recv");
                    exit(1);
                }

                if ((i % PROGRESS) == 0)
                    printf("%d packets sent and received\n", i);
                break;
            }
        }
        sleep(1);
        return(0);
    }
```

SEE ALSO

atm_intro(4N)

NAME

atm_client_null − example client ATM program using a null AAL

EXAMPLE

```
/*
 * Copyright (c) 1992 by Fore Systems, Inc.
 */

#ifndef lint
static char _atm_client_null_c_[] = "@(#)$Id: atm_client_null.c,v 1.3 1994/06/28 20:18:38 jshirron Exp $ FSI";
#endif /* lint */

#include <stdio.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <fore_atm/fore_atm_user.h>

#define SERVER_SAP      4096
#define HOST_CELL_SIZE  56      /* ATM Computer Interface cell size */

main(argc, argv)                                                              main
    int     argc;
    char    *argv[];
{
    int fd, cells, max_cells, octets, qlen;
    Atm_info info;
    Atm_endpoint dst;
    Atm_qos qos;
    Atm_qos_sel qos_selected;
    Atm_sap ssap;
    Aal_type aal = aal_null; /* specify a null connection */
    Atm_dataflow dataflow = simplex;
    char *device_name;
    char *tbuf;
    extern char *malloc();

    if (argc < 3) {
        fprintf(stderr, "Usage: %s device server−hostname [server−sap]\n",
                    argv[0]);
        exit(1);
    }

    device_name = malloc(strlen("/dev/") + strlen(argv[1]) + 1);
    sprintf(device_name, "/dev/%s", argv[1]);

    if ((fd = atm_open(device_name, O_RDWR, &info)) < 0) {
        perror("atm_open");
        exit(1);
    }
    tbuf = malloc(info.mtu);
    max_cells = info.mtu / HOST_CELL_SIZE; /* Max number of cells */

    /*
     * Let ATM driver assign a source SAP by specifying
     * sap of zero. Set incoming connect request length to
     * zero since we´re the client. Servers set queue length
     * to non zero.
     */
    ssap = 0;
    qlen = 0;
    if (atm_bind(fd, ssap, &ssap, qlen) < 0) {
        atm_error("atm_bind");
        exit(1);
    }
    printf("SAP assigned=%d\n", ssap);
```

```
        /*
         * Initialize the server´s data link address.
         */
        if (atm_gethostbyname(argv[2], &dst.nsap) < 0) {
            fprintf(stderr, "atm_gethostbyname failed\n");
            exit(1);
        }
        dst.asap = (argc == 4) ? atoi(argv[3]) : SERVER_SAP;

        /*
         * Request some quality of service.
         */
        qos.peak_bandwidth.target = 256; /* kbit/sec */
        qos.peak_bandwidth.minimum = 128; /* kbit/sec */
        qos.mean_bandwidth.target = 128; /* kbit/sec */
        qos.mean_bandwidth.minimum = 64; /* kbit/sec */
        qos.mean_burst.target = 2;   /* 2 kbit packet length */
        qos.mean_burst.minimum = 1; /* 1 kbit packet length */

        if (atm_connect(fd, &dst, &qos, &qos_selected, aal, dataflow) < 0) {
            atm_error("atm_connect");
            exit(1);
        }
        printf("selected qos peak=%d, mean=%d, burst=%d\n",
            qos_selected.peak_bandwidth,
            qos_selected.mean_bandwidth,
            qos_selected.mean_burst);

        for (cells = 1; cells <= max_cells; cells++) {

            octets = cells * HOST_CELL_SIZE;

            /*
             * The format of cells in `tbuf´ is defined in
             * the appropriate ATM Computer Interface User´s
             * Manual. The 48 octet cell payload is defined
             * by the application.
             */

            if (atm_send(fd, tbuf, octets) < 0) {
                atm_error("atm_send");
                exit(1);
            }
        }
        return(0);
    }
```

SEE ALSO

atm_intro(4N)

NAME

atm_close − close a connection with an ATM endpoint

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

int atm_close(fd)
int fd;

DESCRIPTION

atm_close( ) closes an ATM connection associated with *fd.* In the case of a simplex or duplex connection, **atm_close( )** will close the remote side of the connection as well as closing the connection locally. On SunOS and IRIX platforms, the Stream associated with *fd* will be dismantled as well.

In the case of a multicast connection, the behavior depends on the role of the caller. If the sender calls **atm_close( ),** all destination branches of the connection will be closed. If one of the destinations calls **atm_close( ),** only its branch of the multicast connection will be closed; the remaining destinations will be unaffected. The sender's connection will be closed when the last destination calls **atm_close( ).**

RETURN VALUES

atm_close( ) returns the result of the **close**(2) system call.

ERRORS

Error conditions are returned from the **close**(2) system call.

SEE ALSO

atm_intro(4N)

NAME

atm_connect − establish a connection with another ATM end-point

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

atm_connect(fd, dest, qos, sel, aal, dataflow)
int fd;
Atm_endpoint *dest;
Atm_qos *qos;
Atm_qos_sel *sel;
Aal_type aal;
Atm_dataflow dataflow;

DESCRIPTION

atm_connect( ) is used to establish a connection with a remote ATM end-point. *fd* identifies the file descriptor over which communication will be established.

dest specifies the destination ATM end-point address.

qos (Quality Of Service) argument specifies the caller's requested target and minimum acceptable network bandwidth requirements. The ATM network uses this information to allocate network bandwidth during connection establishment.

The *sel* argument is returned to the caller and specifies the bandwidth resources actually allocated for the connection. The target bandwidth values may have been adjusted downward by the network.

The *aal* argument specifies the ATM Adaptation Layer (AAL) to be used for data exchange. Three AAL types are supported: AAL 3/4, AAL 5, and a null AAL. The AAL 3/4 and AAL 5 types allow the application to send and receive data with the ATM device driver handling all segmentation of the data into ATM cells and the reassembly of ATM cells back into application data. The null AAL allows applications to exchange ATM cells. Cells buffered for transmission and reception are encoded as per FORE Systems' ATM Computer Interface cell format (see the appropriate ATM Computer Interface User's Manual for details).

The *dataflow* argument is used to specify simplex, duplex, or multicast communication. When simplex communication is selected the caller blocks until an ATM connection is established to the destination. If a duplex connection is requested the caller blocks until an ATM connection is established in both directions. If a multicast connection is specified the caller blocks until the destination has been added to the unidirectional multicast connection tree rooted at the sender. Subsequent calls to **atm_connect( )** can be used to add additional destinations. Applications must agree on simplex, duplex, or multicast communication beforehand.

See **atm_client**(4N) for usage of **atm_connect( ).**

RETURN VALUES

atm_connect( ) returns 0 on success. On failure, it returns -1 and sets **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors returned by **atm_connect( ).**

ERRORS

DL_NOTINIT	The network has not been initialized by the ATM switch.
DL_OUTSTATE	Connection already exists for the file descriptor or file descriptor not bound.
DL_UNSUPPORTED	An attempt was made to establish an AAL NULL connection with an SBA-200 involved (currently unsupported).
ECONNREFUSED	Connection failed because the destination ATM end-point did not respond. Will also fail if the minimum requested bandwidth could not be allocated.

atm_connect( ) also can fail with errors returned from the **getmsg**(2) and **putmsg**(2) system calls on the SunOS and IRIX platforms.

On ULTRIX platforms, **atm_connect( )** on a closed connection may cause a SIGPIPE signal to be sent to the calling process. Unless caught or ignored, this terminates the process.

SEE ALSO

atm_intro(4N)

NAME

atm_error − print an ATM error message

SYNOPSIS

extern int atm_errno;

void atm_error(errormsg)
char *errormsg;

DESCRIPTION

atm_error( ) prints a message to standard error which describes the last error received during an ATM user-level library routine. As with **perror**(3) the user-supplied *errormsg* is printed followed by a colon and a textual description of **atm_errno.**

SEE ALSO

atm_intro(4N), **perror**(3)

NAME

atm_gethostbyname – get the ATM address for a hostname

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

atm_gethostbyname(hostname, dest_addr)
char *hostname;
Atm_address *dest_addr;

DESCRIPTION

atm_gethostbyname( ) finds the ATM address for *hostname* by first using **gethostbyname**(3N) to determine the IP address for *hostname,* and then looking for that address in the ATM device driver's ARP table. If no entry is found, an attempt is made to contact *hostname* using IP over ATM, to force an ARP table entry to be created.

RETURN VALUES

atm_gethostbyname( ) returns 0 on success and places the ATM address in *dest_addr.* On failure, it returns -1 and sets **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors returned by **atm_gethostbyname( ).**

BUGS

atm_gethostbyname( ) only works for hosts that are running IP over ATM.

SEE ALSO

atm_intro(4N)

NAME

atm_intro - introduction to ATM user-level library functions

DESCRIPTION

Fore Systems' user-level ATM library routines provide a portable interface to the ATM data link layer. The library routines for SunOS and IRIX platforms are STREAMS-based and communicate with the ATM device driver using the *Data Link Provider Interface* (OSI Work Group, Unix International). The library routines for ULTRIX platforms are socket-based and communicate with the ATM device driver using the *protocol switch table.* Although the internals differ, the same application programming interface is provided on all platforms.

The ATM library routines provide a connection-oriented client and server model. Before data can be transferred a connection has to be established between client and server. The data transfer phase is a best effort delivery system. End-to-end flow control and retransmission are left to the application.

The ATM library can be used to establish connections between two end-systems in a point-to-point configuration (i.e. in the absence of a switch).

ADDRESSING

End-systems and switches can be identified by name and Network Service Access Point (NSAP). The name is a unique ASCII string. The NSAP is also unique and specifies the ATM-layer network address.

An ATM application is referred to as an ATM end-point. Each ATM end-point has a unique end-point address which is used for establishing communications. An end-point address is comprised of an NSAP and an Application Service Access Point (ASAP). The following structure is used to specify ATM end-points for use in the ATM library routines.

```
typedef u_int Atm_sap;  /* Service Access Point */

struct Atm_address {     /* hardware address */
    char addr[8];
};
typedef struct Atm_address Atm_address;

struct Atm_endpoint {
    Atm_sap      asap;   /* Application Service Access Point */
    Atm_address nsap;    /* Network Service Access Point */
};
typedef struct Atm_endpoint Atm_endpoint;
```

An ASAP is unique for a given end-system and is associated by the ATM device driver with a file desctriptor. An NSAP is comprised of an ATM switch id and a port number on the switch. Each ATM end-system is attached to a particular ATM switch port.

The NSAP is used by the ASX switch control daemon (see **asxd**(8C)) to route connection requests and responses through the network. An NSAP is exchanged between ATM user-level library functions and the application. It typically does not need to be interpreted by the application. For example **atm_gethostbyname**(4N) returns an NSAP which can be passed directly to **atm_connect**(4N) by the application, and **atm_listen**(4N) returns an NSAP which can be passed directly to **atm_accept**(4N).

ASAPs in the range of 0 through SAPS_RESERVED are reserved for use by the ATM device driver. ASAPs starting at SAPS_UNRESERVED are for general use.

CONNECTION ESTABLISHMENT

The ATM device driver uses a signaling protocol for establishing connections to remote ATM end-points. The protocol is defined in *SPANS: Simple Protocol for ATM Network Signaling,* available from Fore Systems, Inc.

Applications first open a file descriptor with **atm_open**(4N) and then bind a local ASAP to the file descriptor with **atm_bind**(4N). The local NSAP is also implicitly bound to the same file descriptor. The remote ASAP and NSAP are associated with the file descriptor when a connect indication or a connect confirmation is received.

Connections are established using **atm_connect**(4N) in combination with **atm_accept**(4N). These operations allow the data transfer to be specified as simplex (data flowing in one direction), duplex (data flowing in both directions), or multicast (data flowing in one direction to multiple destinations).

ATM *Virtual Path Identifiers (VPI)* and *Virtual Channel Identifiers (VCI)* are allocated by the network during connection establishment. The ATM device driver associates the VPI/VCI with an

ASAP which is in turn associated with a file descriptor. When a connection is duplex, both an incoming and an outgoing VPI/VCI are associated with the ASAP; the two need not be the same.

Resources (bandwidth) are reserved for each connection. The *peak bandwidth* is the maximum (burst) rate at which the source produces data, measured in kilobits per second. The *mean bandwidth* is the average bandwidth expected over the lifetime of the connection, also measured in kilobits per second. The *mean burst* length is the average size in kilobits of packets sent at peak bandwidth.

The network uses the bandwidth information in the connection request to reduce the chances of buffer overflow, and will refuse the connection if insufficient resources are available. If the bandwidth is specified as zero, the connection is assumed to carry low-priority, loss-tolerant traffic; such a connection request will not be refused due to lack of intermediate resources.

The following structures are used to specify the bandwidth requirements for use in the ATM library routines.

```
struct Atm_qos_range {
    u_int    target;      /* target quality of service */
    u_int    minimum;     /* minimum acceptable qos */
};
typedef struct Atm_qos_range Atm_qos_range;

struct Atm_qos {
    Atm_qos_range peak_bandwidth;
    Atm_qos_range mean_bandwidth;
    Atm_qos_range mean_burst;
};
typedef struct Atm_qos Atm_qos;
```

Connections can be established when two end-systems are connected in a point-to-point configuration (i.e. without a switch). In the point-to-point configuration only the ASAP part of ATM endpoint address is used to establish connections. The NSAP is ignored.

DATA TRANSFER

Applications choose the ATM Adaptation Layer (AAL) to be used for data exchange. The following structure is used to specify the AAL for use in the ATM library routines. AAL types 1 and 2 are not currently supported, and types 3 and 4 are treated identically.

```
enum Aal_type { /* Atm Adaptation Layer types */
  aal_null,
  aal_type_1,    /* circuit emulation */
  aal_type_2,    /* variable bit rate video */
  aal_type_3,    /* connection-oriented data */
  aal_type_4,    /* connectionless data */
  aal_type_5,    /* connection-oriented data */
};
typedef enum Aal_type Aal_type;
```

AAL 3/4 and AAL 5 allow the application to send and receive data with the ATM device driver handling all segmentation of the data into ATM cells and the reassembly of ATM cells back into application data.

The null AAL allows applications to exchange ATM cells. With this interface other AALs can be implemented by user programs. Cells buffered for transmission and reception are encoded as per Fore Systems' ATM Computer Interface cell format (see the appropriate ATM Computer Interface User's Manual for details).

With any given AAL, data transfer can be simplex, duplex, or multicast. Applications statically agree on the data flow type. The following structure is used to specify data flow for use in the ATM library routines.

```
enum Atm_dataflow {
    simplex,          /* unidirectional data flow */
    duplex,           /* bidirectional data flow */
    multicast,        /* multicast data flow */
};
typedef enum Atm_dataflow Atm_dataflow;
```

FILES

/dev/fa*

.../fore/lib/libatm.a
.../fore/include/fore_atm/fore_atm_user.h

SEE ALSO

atm_open(4N), **atm_bind**(4N), **atm_listen**(4N), **atm_accept**(4N), **atm_connect**(4N), **atm_send**(4N), **atm_recv**(4N), **atm_close**(4N), **atm_client**(4N), **atm_server**(4N), **atm_error**(4N), **atm_setbatchsize**(4N), **atm_getbatchsize**(4N), **atm_intro**(8C), **asxd**(8C)

NAME

atm_listen − listen for an ATM connection request

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

atm_listen(fd, conn_id, calling, qos, aal)
int fd;
int *conn_id;
Atm_endpoint *calling;
Atm_qos *qos;
Aal_type *aal;

DESCRIPTION

atm_listen( ) listens for a connect request from a remote ATM end-point. *fd* identifies the listening file descriptor.

On return *conn_id* contains a connection id to be used with **atm_accept**(4N). *conn_id* uniquely identifies the connect indication. *conn_id* enables the user to listen for multiple connect requests and then respond to them in any order.

On return *calling* contains the remote ATM end-point address.

On return *qos* (Quality Of Service) specifies the remote user's requested quality of service including target and minimum acceptable network bandwidth requirements. The target bandwidth values may have been adjusted downward by the network.

On return *aal* specifies the ATM Adaptation Layer (AAL) to be used for data exchange. Three AAL types are supported: AAL 3/4, AAL 5, and a null AAL. The AAL 3/4 and AAL 5 types allows the application to send and receive data with the ATM device driver handling all segmentation of the data into ATM cells and the reassembly of ATM cells back into application data. The null AAL allows applications to exchange ATM cells. Cells buffered for transmission and reception are encoded as per Fore Systems' ATM Computer Interface cell format (see the appropriate ATM Computer Interface User's Manual for details).

See **atm_server**(4N) for usage of **atm_listen( ).**

RETURN VALUES

atm_listen( ) returns 0 on success. On failure, it returns -1 and sets **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors returned by **atm_listen( ).**

ERRORS

atm_listen( ) can fail with errors returned from the **getmsg**(2) and **putmsg**(2) system calls on the SunOS and IRIX platforms.

NOTES

The concurrent server model is not supported in this software release. However *conn_id* is still returned from **atm_listen( )** and required in the call to **atm_accept**(4N).

SEE ALSO

atm_intro(4N)

NAME

atm_open − open an ATM connection endpoint and get ATM connection information

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

```
atm_open(dev, flags, info)
char *dev;
int flags;
Atm_info *info;
```

DESCRIPTION

atm_open( ) attempts to open the device specified by *dev* using the open *flags* specified. These arguments are passed unmodified to the **open**(2) system call. **atm_open( )** returns a file descriptor for use in additional ATM library calls. If the *info* argument is non-zero, information about the ATM link is returned in the structure it points to. The Atm_info struct is defined as

```
struct Atm_info {
        int mtu;        /* Maximum Transmission Unit */
};
typedef struct Atm_info Atm_info;
```

Currently, only the *MTU* (Maximum Transmission Unit) is returned in the *info* parameter. This value indicates the maximum data size, in bytes, that can be sent or received using the **atm_send**(4N) and **atm_recv**(4N) ATM library routines.

RETURN VALUES

atm_open( ) returns a valid file descriptor on success. On failure, it returns -1 and sets **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors returned by **atm_open( ).**

ERRORS

atm_open( ) can fail with errors returned from the **open**(2), **ioctl**(2), system calls. Additionally, errors may be returned from the **getmsg**(2) and **putmsg**(2) system calls on the SunOS and IRIX platforms.

On ULTRIX platforms, **atm_open( )** on a closed connection may cause a SIGPIPE signal to be sent to the calling process. Unless caught or ignored, this terminates the process.

SEE ALSO

atm_intro(4N)

NAME

atm_bind_pvc, atm_unbind_pvc, atm_connect_pvc, atm_disconnect_pvc – ATM PVC operations

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

atm_bind_pvc(fd, vpvc, aal, qos)
int fd;
Vpvc vpvc;
Aal_type aal;
Atm_conn_resource *qos;

atm_unbind_pvc(fd, vpvc)
int fd;
Vpvc vpvc;

atm_connect_pvc(fd, vpvc, aal, qos)
int fd;
Vpvc vpvc;
Aal_type aal;
Atm_conn_resource *qos;

atm_disconnect_pvc(fd, vpvc)
int fd;
Vpvc vpvc;

DESCRIPTION

atm_bind_pvc( ) associates the file descriptor *fd* with the incoming permanent virtual circuit (PVC) identified by the VPI/VCI pair *vpvc.* The resources (bandwidth) specified in *qos* will be reserved for the PVC. Once a call to atm_bind_pvc( ) has been successfully performed, all traffic received on the device associated with *fd* with VPI/VCI pair *vpvc* will be reassembled with AAL type *aal* and made available for receiving via **atm_recv(4N)** and **atm_recvfrom(4N). atm_bind_pvc( )** may be called multiple times (with different *vpvc* values) to associate *fd* with several incoming PVCs.

atm_unbind_pvc( ) disassociates the file descriptor *fd* from the incoming PVC identified by the VPI/VCI pair *vpvc.* Any traffic received on the PVC will no longer be made available for receiving.

atm_connect_pvc() associates the file descriptor *fd* with the outgoing PVC identified by the VPI/VCI pair *vpvc.* The resources (bandwidth) specified in *qos* will be reserved for the PVC. Once a call to **atm_connect_pvc( )** has been successfully performed, **atm_send(4N)** and **atm_sendto(4N)** may be used to transmit traffic from the device associated with *fd* with VPI/VCI pair *vpvc* using AAL type *aal.* **atm_connect_pvc( )** may be called multiple times (with different *vpvc* values) to associate *fd* with several outgoing PVCs.

atm_disconnect_pvc( ) disassociates the file descriptor *fd* from the outgoing PVC identified by the VPI/VCI pair *vpvc.* Traffic can no longer be transmitted on the PVC.

Note that it is not necessary to explicitly call **atm_unbind_pvc(** or **atm_disconnect_pvc( )** for each attached PVC before closing *fd.* **atm_close(4N)** will automatically detach *fd* from all PVCs.

RETURN VALUES

These functions return 0 on success. On failure, they return -1 and set **atm_errno** to indicate the error. **atm_error(4N)** can be used to print a text description of any errors returned.

ERRORS

These functions can also fail with errors returned from the **ioctl(2)** system call.

SEE ALSO

atm_close(4N), atm_error(4N), atm_intro(4N), atm_recv(4N), atm_recvfrom(4N), atm_send(4N), atm_sendto(4N), atm_error(4N), ioctl(2)

NAME

atm_recv, atm_recvfrom, atm_send_null − receive data from an established ATM connection

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

```
atm_recv(fd, buf, len)
int fd;
caddr_t buf;
int len;
```

```
atm_recvfrom(fd, buf, len, vpvc)
int fd;
caddr_t buf;
int len;
Vpvc *vpvc;
```

```
atm_recv_null(fd, buf, len, hdr)
int fd;
caddr_t buf;
int len;
Atm_hdr *hdr;
```

DESCRIPTION

The **atm_recv( )** family of functions is used to receive data from an established ATM connection. **atm_recv( )** is used with switched virtual circuits (SVCs) and may be used with permanent virtual circuits (PVCs) only if *fd* is associated with a single PVC (see **atm_connect_pvc(4N)),** while **atm_recvfrom( )** may only be used with PVCs. If *vpvc* is not a NULL pointer, the VPI/VCI of the connection the data was received on may be filled in. For a given atm_recvfrom(call, VPI/VCI information is only returned if the data returned by that call is from the start of a message. Otherwise, the location pointed to by *vpvc* is not changed.

fd identifies the connection (or connections when *fd* is associated with multiple incoming PVCs) from which to receive the data. *buf* identifies the buffer to hold the data to be received. *len* specifies the maximum amount of data to be received. This length must not be larger than the size of *buf.*

It is important to note that the amount of data received may be less than *len.* Therefore, it may be necessary to invoke **atm_recv( )** multiple times to read all data. If the connection uses *AAL* type *aal_null,* it is possible to control the amount of data that is buffered in the driver before the **atm_recv( )** family returns. See **atm_setbatchsize**(4N), **atm_getbatchsize**(4N), and **atm_connect**(4N) for details.

When receiving data on a 100 series adapter over connections with an **AAL** type of *aal_null,* atm_recv(and atm_recvfrom(must be used. Each message will contain some number of ATM cells presented in FORE's intermediate cell format (see atm_intro(4N) for details on cell formats). Therefore each message will always be a multiple of 56 bytes long.

When receiving data on a 200 series adapter over connections with an **AAL** type of *aal_null,* atm_recv_null(must be used. Each message will contain some number of complete cell payloads, all received with the same ATM cell header. Therefore, each message will be a multiple of 48 bytes long. If *hdr* is not a NULL pointer, the first four bytes of the ATM header may be returned. For a given atm_recv_null(call, header information is only returned if the data returned by that call is from the start of a message. Otherwise, the location pointed to by *hdr* is not changed. Note that any change in a received cell header for a given connection will cause the atm_recv() family to return all previously buffered payloads immediately, independent of the batch size in effect at the time.

RETURN VALUES

The **atm_recv( )** family returns the number of bytes received on success. On failure, they return -1 and set **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors.

ERRORS

The **atm_recv( )** family can also fail with errors returned from the **read**(2) and **readv**(2) system calls.

SEE ALSO

atm_connect(4N), **atm_error**(4N), **atm_intro**(4N), **atm_pvc**(4N), **atm_getbatchsize**(4N), **atm_setbatchsize**(4N), **read**(2), **readv**(2)

NAME

atm_send, atm_sendto, atm_send_null − send data over an established ATM connection

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

```
atm_send(fd, buf, len)
int fd;
caddr_t buf;
int len;

atm_sendto(fd, buf, len, vpvc)
int fd;
caddr_t buf;
int len;
Vpvc vpvc;

atm_send_null(fd, buf, len, hdr)
int fd;
caddr_t buf;
int len;
Atm_hdr hdr;
```

DESCRIPTION

The **atm_send( )** family of functions attempts to write *len* bytes of data from the buffer pointed to by *buf* over an established ATM connection referenced by the file descriptor *fd.* **atm_send( )** is used with switched virtual circuits (SVCs) and may be used with permanent virtual circuits (PVCs) only if *fd* is associated with a single PVC (see **atm_connect_pvc(4N)).**

When *fd* is associated with multiple PVCs, **atm_sendto(** is used to transmit the data over the connection identified by the VPI/VCI pair *vpvc.* The identifier *vpvc* must correspond to an outgoing PVC actively connected from a previous call to **atm_connect_pvc(4N). atm_sendto( )** may not be used with SVCs.

When sending data on a 100 series adapter over connections with an *AAL* type of *aal_null,* atm_send() and atm_sendto() must be used. *buf* specifies a sequence of cells in FORE's intermediate cell format (see atm_intro(4N) for details on cell formats). *len* must correspond to a number of complete cells, so it must be a multiple of 56.

When sending data on a 200 series adapter over connections with an *AAL* type of *aal_null,* atm_send_null() must be used. The first four bytes of the ATM header are specified in *hdr,* while the HEC is automatically computed and inserted by the hardware. *buf* specifies a sequence of 48 byte cell payloads, each to be transmitted with the specified header. *len* must correspond to a number of complete payloads, so it must be a multiple of 48. Note that atm_send_null(is used when *fd* is associated with an SVC as well as with multiple PVCs. When *fd* references an SVC, the VPI/VCI pair, as specified in the ATM cell header *hdr,* must be zero. When *fd* references a PVC, the VPI/VCI must correspond to an outgoing PVC actively connected from a previous call to atm_connect_pvc(4N).

The number of bytes to send as specified by *len* must not exceed the size of *buf,* nor must the number of bytes exceed the *maximum transmission unit (MTU)* for the connection. See **atm_open**(4N) for details on the *MTU.* In addition, for connections using an *AAL* type of *aal_null, len* must follow the restrictions outlined above.

RETURN VALUES

The **atm_send( )** family returns the number of bytes transmitted on success. On failure, they return -1 and set **atm_errno** to indicate the error. **atm_error(4N)** can be used to print a text description of errors.

ERRORS

The **atm_send( )** family can also fail with errors returned from the **write(2)** and **writev(2)** system calls.

On ULTRIX platforms, calling **atm_send( )** or **atm_sendto( )** on a closed connection may cause a SIGPIPE signal to be sent to the calling process. Unless caught or ignored, this terminates the process.

SEE ALSO

atm_intro(4N), atm_open(4N), atm_pvc(4N), write(2), writev(2)

NAME

atm_server_aal4 − example server ATM program using AAL 3/4

EXAMPLE

```
/*
 * Copyright (c) 1992 by Fore Systems, Inc.
 */

#ifndef lint
static char _atm_server_aal4_c_[] = "@(#)$Id: atm_server_aal4.c,v 1.3 1994/06/28 20:18:39 jshirron Exp $ FSI";
#endif /* lint */

#include <stdio.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <fore_atm/fore_atm_user.h>

#define SERVER_SAP   4096
#define PROGRESS     1024      /* report progress every n packets */

main(argc, argv)                                                    main
    int       argc;
    char      *argv[];
{
    int fd, i, mtu, qlen, conn_id;
            u_int switchid, portid;
    Atm_info info;
    Atm_endpoint calling;
    Atm_qos qos;
    Atm_sap ssap;
    Aal_type aal;
    Atm_dataflow dataflow = simplex;
    char *device_name;
    char *tbuf, *rbuf;
    extern char *malloc();

    if (argc < 2) {
        fprintf(stderr, "Usage: %s device [server-sap]\n",
                        argv[0]);
        exit(1);
    }

    device_name = malloc(strlen("/dev/") + strlen(argv[1]) + 1);
    sprintf(device_name, "/dev/%s", argv[1]);

    if ((fd = atm_open(device_name, O_RDWR, &info)) < 0) {
        perror("atm_open");
        exit(1);
    }
    mtu = info.mtu; /* Max packet size */
    tbuf = malloc(mtu);
    rbuf = malloc(mtu);

    /*
     * Bind to a well known sap and set pending
     * connect request queue length.
     */
    ssap = (argc == 3) ? atoi(argv[2]) : SERVER_SAP;
    qlen = 1;
    if (atm_bind(fd, ssap, &ssap, qlen) < 0) {
        atm_error("atm_bind");
        exit(1);
    }
    printf("SAP assigned=%d\n", ssap);
```

```
/*
 * Wait for a connection request.
 */
if (atm_listen(fd, &conn_id, &calling, &qos, &aal) < 0) {
	atm_error("atm_connect");
	exit(1);
}
/*
 * Extract the switch id and port id from the NSAP.
 */
GET_SWITCH(switchid, calling.nsap);
GET_PORT(portid, calling.nsap);

printf("calling switch=%d, port=%d, sap=%d, aal=%d\n",
	switchid, portid, calling.asap, aal);

printf("qos target peak=%d, mean=%d, burst=%d\n",
	qos.peak_bandwidth.target,
	qos.mean_bandwidth.target,
	qos.mean_burst.target);

printf("qos minimum peak=%d, mean=%d, burst=%d\n",
	qos.peak_bandwidth.minimum,
	qos.mean_bandwidth.minimum,
	qos.mean_burst.minimum);

printf("connect conn_id=%d\n", conn_id);

/*
 * Request some quality of service.
 */
qos.peak_bandwidth.target = 0; /* kbit/sec */
qos.peak_bandwidth.minimum = 0; /* kbit/sec */
qos.mean_bandwidth.target = 128; /* kbit/sec */
qos.mean_bandwidth.minimum = 64; /* kbit/sec */
qos.mean_burst.target = 2;  /* 2 kbit packet length */
qos.mean_burst.minimum = 1; /* 1 kbit packet length */

/*
 * Accept the connection request.
 */
if (atm_accept(fd, fd, conn_id, &qos, dataflow) < 0) {
	atm_error("atm_accept");
	exit(1);
}

for (i = 1; i <= mtu; i++) {

	if (atm_recv(fd, rbuf, i) < 0) {
		atm_error("atm_recv");
		break;
	}

	switch (dataflow) {
	case simplex:

		if ((i % PROGRESS) == 0)
			printf("%d packets received\n", i);
		break;

	case duplex:

		if (atm_send(fd, tbuf, i) < 0) {
			atm_error("atm_send");
			break;
```

```
                }

                if ((i % PROGRESS) == 0)
                    printf("%d packets received and sent\n", i);
                break;
            }
        }
        printf("%d packets sent and/or received\n", i);
        sleep(1);
        return(0);
    }
```

SEE ALSO

atm_intro(4N)

NAME

atm_server_null − example server ATM program using a null AAL

EXAMPLE

```
/*
 * Copyright (c) 1992 by Fore Systems, Inc.
 */

#ifndef lint
static char _atm_server_null_c_[] = "@(#)$Id: atm_server_null.c,v 1.3 1994/06/28 20:18:40 jshirron Exp $ FSI";
#endif /* lint */

#include <stdio.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <fore_atm/fore_atm_user.h>

#define SERVER_SAP      4096
#define HOST_CELL_SIZE  56      /* ATM Computer Interface cell size */

main(argc, argv)
    int     argc;
    char    *argv[];
{
    int fd, cells, mtu, octets,  qlen, conn_id;
    u_int switchid, portid;
    Atm_info info;
    Atm_endpoint calling;
    Atm_qos qos;
    Atm_sap ssap;
    Aal_type aal;
    Atm_dataflow dataflow = simplex;
    char *device_name;
    char *rbuf;
    extern char *malloc();

    if (argc < 2) {
        fprintf(stderr, "Usage: %s device [server-sap]\n",
                    argv[0]);
        exit(1);
    }

    device_name = malloc(strlen("/dev/") + strlen(argv[1]) + 1);
    sprintf(device_name, "/dev/%s", argv[1]);

    if ((fd = atm_open(device_name, O_RDWR, &info)) < 0) {
        perror("atm_open");
        exit(1);
    }
    mtu = info.mtu; /* Max transmission unit */
    rbuf = malloc(info.mtu);

    /*
     * Bind to a well known sap and set pending
     * connect request queue length.
     */
    ssap = (argc == 3) ? atoi(argv[2]) : SERVER_SAP;
    qlen = 1;
    if (atm_bind(fd, ssap, &ssap, qlen) < 0) {
        atm_error("atm_bind");
        exit(1);
    }
    printf("SAP assigned=%d\n", ssap);
```

main

```
/*
 * Wait for a connection request.
 */
if (atm_listen(fd, &conn_id, &calling, &qos, &aal) < 0) {
     atm_error("atm_connect");
     exit(1);
}
/*
 * Extract the switch id and port id from the NSAP.
 */
GET_SWITCH(switchid, calling.nsap);
GET_PORT(portid, calling.nsap);

printf("calling switch=%d, port=%d, sap=%d, aal=%d\n",
     switchid, portid, calling.asap, aal);

printf("qos target peak=%d, mean=%d, burst=%d\n",
     qos.peak_bandwidth.target,
     qos.mean_bandwidth.target,
     qos.mean_burst.target);

printf("qos minimum peak=%d, mean=%d, burst=%d\n",
     qos.peak_bandwidth.minimum,
     qos.mean_bandwidth.minimum,
     qos.mean_burst.minimum);

printf("connect conn_id=%d\n", conn_id);

/*
 * Request some quality of service.
 */
qos.peak_bandwidth.target = 256; /* kbit/sec */
qos.peak_bandwidth.minimum = 128; /* kbit/sec */
qos.mean_bandwidth.target = 128; /* kbit/sec */
qos.mean_bandwidth.minimum = 64; /* kbit/sec */
qos.mean_burst.target = 2;  /* 2 kbit packet length */
qos.mean_burst.minimum = 1; /* 1 kbit packet length */

/*
 * Accept the connection request.
 */
if (atm_accept(fd, fd, conn_id, &qos, dataflow) < 0) {
     atm_error("atm_accept");
     exit(1);
}

/*
 * Set batch size to 32. Causes the ATM device
        * driver to send cells upstream in batchs of 32.
 * The default batch size is 1.
 */
if (atm_setbatchsize(fd, 32) < 0) {
     atm_error("atm_setbatchsize");
     exit(1);
}

for(cells = 0; ;) {

     if ((octets = atm_recv(fd, rbuf, mtu)) < 0) {
          atm_error("atm_recv");
          break;
     }
     cells += (octets / HOST_CELL_SIZE);
```

```
            /*
             * The format of cells in `rbuf´ is defined in
             * the appropriate ATM Computer Interface User´s
             * Manual. The 48 octet cell payload is defined
             * by the application.
             */
        }
        printf("%d cells received\n", cells);
        return(0);
    }
```

SEE ALSO

atm_intro(4N)

NAME

atm_setbatchsize, atm_getbatchsize – set/get batch size for null AAL interface

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

int atm_setbatchsize(fd, batch_size)
int fd;
int batch_size;

int atm_getbatchsize(fd, batch_size)
int fd;
int *batch_size;

DESCRIPTION

atm_setbatchsize( ) is used to set the minimum number of cells that must be received by the connection specified by *fd* before **atm_recv**(4N) will return. Note that for the 200 series adapters, any change in the ATM cell header will trigger a return. This interface only applies to ATM connections established with an *AAL* of type *aal_null.* When several PVCs are associated with the same descriptor, the same size applies to each PVC. By default, the batch size is set to *1,* which causes **atm_recv**(4N) to return each cell received for the incoming connection specified by *fd.* A different size can be specified using the *batch_size* argument to **atm_setbatchsize( ).** Setting the batch size to a value other than 1 can improve performance by reducing the overhead incurred per cell. The value must be chosen carefully since a batch size other than 1 will prevent data from moving from the physical level to the ATM level until the number of cells received for the connection is equal to or greater than the batch size. In addition, care must be taken to ensure that the data size, in bytes (56 bytes per cell for 100 series adapters, 48 bytes per cell for 200 series adapters), is not greater than the *Maximum Transmission Unit (MTU)* for the connection. The *MTU* for the link is returned by the **atm_open**(4N) operation.

atm_getbatchsize( ) returns the current value of the batch size in the *batch_size* argument.

RETURN VALUES

atm_setbatchsize( ) and **atm_getbatchsize( )** return 0 on success. On failure, they return -1 and set **atm_errno** to indicate the error. **atm_error**(4N) can be used to print a text description of errors of returned by **atm_setbatchsize( )** and **atm_getbatchsize( ).**

ERRORS

atm_setbatchsize( ) and **atm_getbatchsize( )** can also fail with errors returned from the **ioctl**(2) system call.

SEE ALSO

atm_intro(4N)

NAME

atmio − ATM device control operations

SYNOPSIS

#include <fore_atm/fore_atm_user.h>

DESCRIPTION

The ATM device responds to a number of **ioctl**(2) requests, defined in *<fore_atm/fore_atm_user.h>*. The **ioctl**(2) requests and commands are STREAMS-based on SunOS and IRIX platforms; see **streamio**(4) for details on the STREAMS **ioctl**(2) interface.

This interface is not intended for general use. Use of these operations may interfere with normal use of the ATM network. This interface is likely to change in future releases of the ATM device driver.

IOCTLS FOR THE ATM LAYER

The following **ioctl** commands are applicable to the ATM layer.

ATMIOC_GET_MY_ADDRESS
Returns the ATM-layer address of this ATM device. The result is returned in an *Atm_address* structure, as defined in *<fore_atm/fore_atm_user.h>*.

ATMIOC_FLUSH_ARP
Flushes the internal ATM ARP table for the device.

ATMIOC_DELETE_ARP
Deletes the ARP table entry for the hostname provided as an argument in the form of the *struct arpreq* found in *<net/if_arp.h>*. **ATMIOC_DELETE_ARP** will fail if the given hostname does not exist, or if an address family other than *AF_INET* is specified.

ATMIOC_GET_ARP
Retrieves the ARP information for the hostname specified in the argument of the form *struct arpreq* found in *<net/if_arp.h>*. **ATMIOC_GET_ARP** will fail if the given hostname does not exist, or if an address family other than *AF_INET* is specified.

ATMIOC_SOFT_RESET
Resets the connection information for the ATM device. This can be used to reset the connection information for devices connected to an ATM network if fiber pairs on the switch are physically moved from one port to a separate port.

ATMIOC_SET_BATCH_SIZE
Sets the number of cells to be received for a *null AAL* connection before the data will be returned in the ATM library call **atm_recv**(4N). This allows a the driver to queue up cells to pass up to the user, thereby reducing the overhead incurred for each **atm_recv**(4N) call. Please see **atm_recv**(4N) and **atm_connect**(4N) for details.

ATMIOC_GET_BATCH_SIZE
Returns the current batch size setting.

IOCTLS FOR ATM STATISTICS

The following **ioctl** commands are applicable to the ATM layer and AAL layers.

ATMIOC_ATM_LAYER_STATS
Returns driver statistics for the ATM layer. The statistics are returned in the *Atm_layer_stats* structure defined in *<fore_atm/fore_atm_user.h>*.

ATMIOC_AAL0_STATS
Returns driver statistics for the null adaptation layer. The statistics are returned in the *Aal0_stats* structure defined in *<fore_atm/fore_atm_user.h>*.

ATMIOC_AAL4_STATS
Returns driver statistics for AAL 3/4. The statistics are returned in the *Aal4_stats* structure defined in *<fore_atm/fore_atm_user.h>*.

ATMIOC_AAL5_STATS
Returns driver statistics for AAL 5. The statistics are returned in the *Aal5_stats* structure defined in *<fore_atm/fore_atm_user.h>*.

ATMIOC_ZERO_STATISTICS
Resets all driver statistics to zero.

IOCTLS FOR LOW-LEVEL HARDWARE ACCESS

The following operations support low-level access to the ATM device. They are not for general use.

ATMIOC_INTR_OFF
Turns off the interrupts in the driver for the ATM device. Note that while the hardware interrupts may still occur, they will be caught and then ignored by the driver.

ATMIOC_INTR_ON
Turns on the interrupts that were turned off by *ATMIOC_INTR_OFF.*

ATMIOC_SET_RAW_CELL
Delivers the raw cell as specified as an array of 15 words to the hardware. Note that this raw delivery mechanism should be used with extreme caution as these words are passed to the hardware unchanged and may not be in a format that can be interpreted by the driver code. Cells written using this interface should be read using the *ATMIOC_GET_RAW_CELL* command. Also, interrupts at the driver level should be disabled using *ATMIOC_INTR_OFF* to prevent the driver from attempting to read the cell. Note also that 15 words should be specified to this interface. If the interface is the ASX switch, the first word is interpreted as a route word, otherwise, the first word is simply discarded and the remaining 14 words are delivered. **ATMIOC_SET_RAW_CELL** will fail if the number of words specified is not 15.

ATMIOC_GET_RAW_CELL
Reads a raw cell into an array of 15 words. Note that the first word is reserved for the *route* word for the ASX switch; it is undefined for all other interfaces. Therefore, the actual cell begins at the second word of the 15 words.

IOCTLS FOR ASX SWITCH CONTROL SOFTWARE

The following interfaces are provided to support the Switch Control Software for the ASX switch. These interfaces are not for general use.

ATMIOC_ATTACH_RX_VCI
Associates the specified VCI with this file descriptor. After performing this operation, PDUs arriving over the specified VCI are prefixed with the VCI and delivered via the file descriptor. PDUs that are transmitted on the file descriptor must also be prefixed with a local VCI value. This value will be translated by the switch hardware to determine the outgoing port(s) and outgoing VCI for that PDU.

ATMIOC_DETACH_RX_VCI
Deletes the association of the given VCI with this file descriptor.

IOCTLS FOR ASX SWITCH CONTROLLER IOPORT

The following interfaces support the ASX switch controller ioport. This controller ioport behaves as an ATM computer interface, generally speaking.

ATMIOC_CP_ENAB_LOOPBACK
Enables hardware loopback for the controller ioport. Cells written to the ioport transmit interface will immediately return to the ioport receive interface. Routing of cells to and from the controller ioport is typically done by the switch control software, thus this interface should not be used when the switch control software is running.

ATMIOC_CP_DIS_LOOPBACK
Disables the hardware loopback mechanism for the controller ioport that may have been previously enabled with the *ATMIOC_CP_ENAB_LOOPBACK* interface.

ATMIOC_IOPORT_OFF
Disables the controller ioport completely.

ATMIOC_IOPORT_ON
Enables the controller ioport that may have been disabled using the *ATMIOC_IOPORT_OFF* interface.

IOCTLS FOR PERMANENT VIRTUAL CONNECTIONS

The following **ioctl** commands are applicable to routing of IP traffic over PVCs.

ATMIOC_CREATE_IPPVC
Establishes an association between an IP destination address and a PVC. The IP address and VC identifiers are specified by an *Ip_pvc* structure, defined in *<fore_atm/fore_atm_arp.h>*.

ATMIOC_REMOVE_IPPVC
Removes the association between an IP destination address and a PVC.

ATMIOC_SPANS_OFF
Disables SPANS protocol messages over this ATM link, for use when communicating to a non-SPANS ATM device.

ATMIOC_SPANS_ON
Enables SPANS protocol messages over this ATM link if they have been disabled by *ATMIOC_SPANS_OFF.*

EXAMPLE FOR SUNOS AND IRIX PLATFORMS

The following is an example of the **ioctl**(2) mechanism on STREAMS-based (SunOS and IRIX) platforms.

```
#include <sys/ioctl.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <fore_atm/fore_atm_user.h>

struct strioctl ioc;
Atm_address addr;

ioc.ic_cmd = ATMIOC_GET_MY_ADDRESS;
ioc.ic_timout = 0;
ioc.ic_len = sizeof(addr);
ioc.ic_dp = (caddr_t) &addr;
if (ioctl(atm_fd, I_STR, (caddr_t) &ioc) != 0)
        perror("ioctl: ATMIOC_GET_MY_ADDRESS);
```

EXAMPLE FOR ULTRIX PLATFORMS

The following is an example of the **ioctl**(2) mechanism on ULTRIX platforms.

```
#include <sys/ioctl.h>
#include <fore_atm/fore_atm_user.h>

Atm_address addr;

if (ioctl(atm_fd, ATMIOC_GET_MY_ADDRESS, (caddr_t) &addr) != 0)
        perror("ioctl: ATMIOC_GET_MY_ADDRESS);
```

SEE ALSO

atm_intro(4N), **streamio**(4)